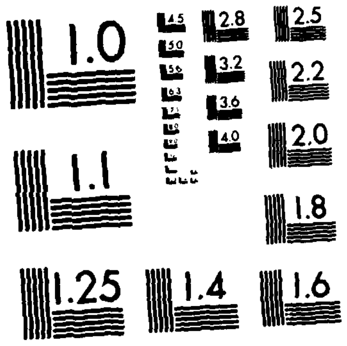
AD-A171 256 A PARALLEL PROCESSOR FOR THE SOLUTION OF LARGE SPARSE SYMMETRIC LINEAR SYSTEMS(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH D A ARPIN 1986 1/4

UNCLASSIFIED AFIT/CI/NR-86-124D F/G 9/2 NL


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

AD-A171 256

1

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/CI/NR 86- 124D	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A Parallel Processor for the Solution of Large, Sparse, Symmetric Linear Systems		5. TYPE OF REPORT & PERIOD COVERED ~~THESIS~~/DISSERTATION
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) David A. Arpin		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS AFIT STUDENT AT: University of Washington		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS AFIT/NR WPAFB OH 45433-6583		12. REPORT DATE 1986
		13. NUMBER OF PAGES 300
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLAS
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

DTIC ELECTE AUG 28 1986 S D B

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

APPROVED FOR PUBLIC RELEASE: IAW AFR 190-1

Lynn Wolaver

LYNN E. WOLAVER 8 Aug 86
Dean for Research and Professional Development
AFIT/NR

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

ATTACHED.

DTIC FILE COPY

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

A Parallel Processor for the Solution of Large, Sparse, Symmetric Linear Systems

by

DAVID A. ARPIN

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

1986

Approved by______________________________________
(Chairperson of Supervisory Committee)

____

____

____

Program Authorized
to Offer Degree____________________________________

Date____

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

The author wishes to express sincere thanks to Professor Yongmin Kim for his guidance in this research and the preparation of this manuscript. In addition, thanks to Professor Loyce Adams, who contributed valuable insights in several areas. Thanks also to Mr. James Moore and Mr. Robert Kaucic, who did a major share of the task of building the prototype of my design.

DEDICATION

This work is dedicated to my wife, Judy, and my children David, Lisa, Alison, and Matt, for their encouragement and support during this pursuit.

University of Washington

Abstract

A PARALLEL PROCESSOR FOR THE SOLUTION OF LARGE, SPARSE SYMMETRIC LINEAR SYSTEMS

by David A. Arpin

Chairperson of the Supervisory Committee:

Professor Yongmin Kim
Department of Electrical Engineering

The need for rapid solution of large, sparse linear systems in certain applications is described. Conventional single processor computers are nearing fundamental speed limits, and will not be able to attain sufficient speeds. Parallel processing allows the application of a large number of processors to these problems, and thus has the potential to achieve faster operation. This dissertation describes a processor designed specifically for this type of problem. This system uses the technique of successive overrelaxation to obtain a solution to the linear system by iteration. The architecture features a separate processor for each variable in the linear system. The processors are arranged in a two dimensional grid, with connections to the four nearest neighbors. Each processor is partitioned into an Arithmetic Unit, which performs the actual computations, and a Communications Unit, to provide the necessary interchange of data between the processors. The design of the processor is described in detail. A detailed simulation is described which shows the performance and efficiency of the proposed system to be very high. A prototype of the processing element has been constructed, and verifies the results obtained in the simulation. Factors which could adversely impact the performance of the system in a full implementation are discussed, and solutions suggested. The architecure is compared with other parallel architecures which could be used for such problems - its advantages and disadvantages are described.

CHAPTER ONE - PROBLEM DESCRIPTION

This dissertation describes the results of my research in developing a computer architecture for the fast solution of large systems of linear equations. This chapter serves as an introduction, giving the motivation for research in this area, describing the context in which the work took place, and outlining the methods used in the research and the presentation of the report. Chapter 2 describes the algorithm selected for use in the system. Chapter 3 describes the architecture I devised to implement this algorithm. The presentation includes a basic description of the system, and also a much more detailed description. The latter description is not crucial to the development of the paper, and may be skipped by the reader who is not interested in this level of detail. Chapter 4 explains the simulation methods used to evaluate the performance of the system, and presents the results of this simulation. Chapter 5 describes the prototype of the system, which was built and tested to demonstrate the basic concepts used in the architecture. Chapter 6 discusses issues in the design and operation of the system not covered in earlier chapters, evaluates the performance of the proposed system, compares this architecture with several other systems which have been proposed or developed, describes areas in need of further investigation, and presents a summary and conclusions.

Sparse Linear System Problems

The initial motivation for this research came from work in the field of impedance imaging [Kim, 1981, 1982, 1984]. This is a non-invasive technique for constructing internal images of objects, including the human body. The technique is being investigated as an alternative to such established techniques as X-rays, ultrasound, and nuclear magnetic resonance imaging for medical applications.

Given the impedance profile of a body and a set of applied voltages and ground points, it is possible to determine the voltage distribution within the body, and from this to compute the current which would enter and leave the body at the electrodes. This is referred to as the "forward problem". Image reconstruction requires the solution of the "inverse problem" - given a set of applied voltages and measured currents (or vice versa), determine the impedance profile of the body. Several algorithms have been developed for this purpose [Woo, 1986]. In general terms, all the proposed algorithms follow the same sequence of steps:

1. Start with an assumed or estimated impedance profile.

2. Place electrodes at selected points on the body. Apply known voltages to these electrodes and measure the currents flowing into the body (or apply fixed

currents and measure the voltages).

3. Solve the forward problem to obtain the voltage distribution within the body.

4. From the voltage distribution and the impedance profile, calculate the currents which should flow into the electrodes. Compare these with the measured currents. If the measured currents are close to the computed ones, the assumed impedance profile can be considered a good representation of the impedance profile of the body, so stop the iteration and present this impedance profile as the completed image. If the measured and computed currents differ, continue with step 5.

5. Adjust the impedance profile in some way to account for the difference between computed and measured currents.

6. Return to step 3 and continue.

The differences in the different algorithms are mainly in how the impedances are adjusted in step 5. The algorithms are iterative, with the image refined for each pass through the loop given by steps 3 through 5 above. The number of iterations required for a satisfactory image depends on many factors, including the resolution desired in the image (i.e., the number and size of elements used in the finite element model), the impedance distribution within the particular object being imaged, the accuracy of the initial guess of the impedance profile, the number of electrodes used, and the criteria used to stop the iteration in step 4. Kim [1982] reported results of image reconstructions for a computer model of a section of the human torso, with different conductivities modeled in various regions. These images required 50 to 200 iterations of the process described above.

A critical portion of these algorithms is the solution of the forward problem in step 3. For all but the simplest impedance profiles, analytic solution of this problem is not feasible. Thus, numerical techniques must be used. The technique used by Kim and others is the finite element method [Huebner and Thornton, 1975]. This is a mathematical technique for solving partial differential equations numerically, by converting the partial differential equation with boundary conditions to a system of linear equations. The region over which the equation is to be solved is divided into small segments, called elements. Selected points within the elements (typically on the corners or edges) are designated as nodes. The variable of interest is calculated only at the discrete node points, rather than throughout the continuous region. The differential equation is discretized over each element, resulting in a set of linear equations for the values of the variables at the nodes contained in that element. These equations are then combined for the entire problem, resulting in a single large linear system of the form:

$$Ax = b \tag{1}$$

where A is a matrix describing the structure of the system and the characteristics of each element; x is a vector of the values of the variable of interest at all the nodes; and b is a vector of the boundary conditions applied to the system.

In the impedance imaging application, matrix A would contain a description of the conductivities or resistivities of the elements which connect the nodes, x would represent the voltages at the nodes, and b would represent the voltages applied or measured at the electrodes (actually, at the nodes to which the electrodes were attached). The structure of this system (i.e., the locations of the nonzero terms in matrix A) needs to be derived only once for a particular application. The values used in matrix A would then represent the estimated impedance profile of the body being measured. The image reconstruction iteration, described above, would solve for vector x in step 3, and adjust the values of the elements of matrix A (but not the structure of the matrix) in step 5.

The linear system resulting from the finite element method, to be solved in step 3, has several characteristics which affect the choice of solution method. The matrix A is sparse, symmetric, and positive definite. Chapter 2 explains why these conditions hold, and their effects on the solution of the system. In general, however, these conditions simplify the solution in important ways. These conditions hold for many problems which are solved by finite element methods in addition to the problem associated with impedance imaging.

One of Kim's concerns in his research was the amount of time required to reconstruct each image. His finite element model contained 1080 nodes, which modeled three thin layers of the torso. He devised a highly efficient implementation of the solution method, and used a fairly powerful minicomputer (TI-990/12, with floating point hardware) for the solution. Still, he found that each iteration through the loop used for image reconstruction required about 20 seconds of computer time, mostly for the solution of the linear system. Thus, images which required 200 iterations would take more than one hour of computer time to construct. In the research environment, such delays are tolerable. However, the impedance imaging technique may prove to have value for clinical applications. This might require significantly larger models, with many more nodes than the small research model used, in order to gain adequate resolution in the resulting images. But the number of arithmetic operations required for the direct solution of linear systems increases with the cube of the number of variables. Thus, a practical image reconstruction system would require significantly faster solution speed for the linear system than Kim observed.

Thus, the technique of impedance imaging presents a need for fast solution of linear systems. This led me to investigate ways of providing more rapid solutions for such problems. There are many techniques for solving linear systems (see Chapter 2) - the selection of a method depends on the characteristics of the particular problem to be solved. For example, techniques have been developed for solving banded and sparse systems to take advantage of the reduced amount of data which must be stored for these systems. Further refinements are possible if the matrix is known to be symmetric. Finally, if the matrix is known to be positive definite, additional simplifications are possible in the solution algorithm. (Chapter 2 contains detailed explanations of these properties.) The impedance imaging technique gives rise to linear systems which are sparse, symmetric, and positive definite, so a solution method could take advantage of these properties.

There are many other applications which give rise to linear systems with these same properties. The finite element method was originally used primarily for structural analysis, and is still used in that field extensively, for the analysis of ever larger and more complex structures [Duff et al., 1986]. Some of the models currently in use have more than 300,000 variables, and require long solution times even on the fastest available computers. More recently, the finite element method has found acceptance for other applications besides structural analysis, in areas such as electromagnetic fields problems [Kim, 1982], heat and fluid flow problems [Hwang and Briggs, 1985], and mathematics [Axelsson and Barker, 1984]. Thus, there are many applications which could benefit from faster solution of the linear systems which arise from the finite element method.

In addition, there are other sources of linear systems besides the finite element method. Another technique for solving partial differential equations is the finite difference method. This method is similar to the finite element method, except the nodes are connected by a very regular grid structure, rather than elements of arbitrary shape. This method is used, for example, in the solution to the Navier-Stokes fluid dynamics equations for atmospheric modeling and weather forecasting. Hwang and Briggs [1985] estimate that a Cray-1 would require 24 hours of computing time to solve these equations and provide a 24 hour weather forecast. Clearly, a faster solution is required for the forecast to be useful. Physicists also use this technique to model systems such as the behavior of the ocean and planetary formation.

Another application of linear systems is in the control of electric power generation and distribution facilities [Weedy, 1979]. In order to control the operation of such systems, it is sometimes necessary to solve for the power flowing in the network - this is known as the load flow problem. Schemes for the automatic control of power systems require very fast solutions of the load flow problem in order to provide timely controls to the system. These systems involve thousands of node values which must be modeled, and therefore require the solution of large linear systems. The matrix in this linear system represents the admittances for the connections between nodes at which the voltage is computed. Since there are typically only a few elements (e.g., generators, transmission lines, or loads), connecting any pair of nodes, the matrix tends to be sparse. Also, the matrices are symmetric for most problems. Thus, this is another application requiring the fast solution of large, sparse linear systems.

Some of these applications of linear systems can tolerate the delays involved in the solution of large linear systems on conventional computers. In a research environment, the researcher can wait hours or days for the solution of a difficult problem to be available. For structural analysis, the time required to analyze a complex structure is small compared to the time required to construct it. In such applications, a faster solution to the problems might be considered desirable, but not essential.

For other applications, however, a faster solution is extremely important. In a power system controller, the response to changing conditions must be fast enough to keep the system in a safe, stable and efficient operational state. In the impedance imaging application, the technique might be used to monitor patients for potentially life threatening conditions. Thus, long delays for image reconstruction would be intolerable. Even in more routine applications, speed can be important. With the level of computing power available to Kim, generation of a single image of adequate resolution for diagnostic purposes could take several hours. If this technique proved useful, there could be a need to perform many such procedures on a routine basis. If each image required excessive amounts af computer time, there would soon be a huge backlog of raw data requiring analysis, and it would take days to get the results of each test. Thus, faster computation would be essential in order to make this a practical diagnostic technique. For structural analysis, a sufficiently fast solution to the linear system, coupled with the graphics and other capabilities of an engineering workstation, could allow a designer to refine and analyze a system on an interactive basis. This could significantly reduce the cost and time required for the design of complex systems.

Parallel Computing

For these, and other similar applications, there would be a significant value in having a machine capable of solving large, sparse linear systems as quickly as possible. Thus, I decided to try to devise a computer architecture which would perform such a function.

The earliest computers were constrained by cost factors to the use of a single processing unit, which would perform all the required functions for a computation, but only one thing at a time. For each instruction, the processor had to fetch the instruction from memory, decode it, fetch the operand, execute the function, and store the result. Then the processor would repeat this cycle for the next instruction. This is referred to as serial computation. While they were slow by today's standards, the early computers were much faster than manual calculations, thus making it possible to perform computations which could not previously be done due to the length of time required. However, the size of the problems of interest kept pace with the increases in speed of computation available, thus requiring ever faster computers.

The technology of computers improved to enhance their speed in two ways. One was to make the serial operations happen faster. The progression of circuitry from mechanical relays through vacuum tubes, transistors, and eventually to high speed and high density integrated circuits; and of memory from punched cards to mechanical magnetic media, ferrite cores, and now advanced semiconductor memories have led to dramatic increases in the speed of operation of computers. Such improvements are highly desirable from the user point of view, since they do not require any changes to the way he uses the computer in order to obtain faster operation. The computer still does the same things it did before, only faster.

Unfortunately, this type of improvement in computer performance can not continue indefinitely. The technology is beginning to approach fundamental physical limits. The switching time of circuits is approaching the time required to propagate the length of the device at the speed of light. This can be offset by making the device smaller, but this approach can not be pursued without limit either - thermal noise effects, the quantum nature of electrical charge, and the wave nature of electrons themselves require devices to be at least a certain size in order to function properly. For example, the proper functioning of a transistor depends on the fact that the flow of charge carrying electrons is controlled by the different energy levels in the depletion region. However, if the device size is too small, many electrons will flow in the wrong direction by tunneling, so the transistor will not work. Uhr [1984] estimates that further improvements in technology will result in devices capable of operating with a basic clock period of 10 to 100 picoseconds, or roughly 1000 times as fast as the fastest machine currently available. At that point, further gains in speed from this approach will be impossible.

The second method of performance improvement results from the reduced cost of hardware. This allows the designer to add hardware to the computer to allow it to perform several tasks at the same time, without adding excessive cost. This is the basic concept of parallel processing. Parallelism can be exploited in many ways:

- The individual steps required to execute instructions can be pipelined (e.g., fetch, decode, execute).

- The execution of complex instructions, such as floating point arithmetic, can be pipelined.

- Special functional units can be added to perform particular tasks, such as DMA channels which provide high speed I/O without intervention by the CPU.

- Interleaved memories allow access to multiple data items at the same time.

- Multiple execution units allow several operations, such as addition, multiplication, and memory access, to occur at the same time.

Some of these are transparent to the user, while others require changes in the way he uses the system. For example, pipelining of overall instruction execution benefits most programs. But pipelining of the execution of floating point operations is effective only if the program succeeds in keeping the pipeline filled with operations. Extra functional units such as DMA controllers or floating point execution units will only benefit applications which have need of the functions these units can provide.

Each of the techniques mentioned above is limited in the amount of improvement it can give to the performance of a computer. Pipelines can only speed up the operation of the computer by a factor of the number of pipeline stages (i.e., approximately 10), and then only if the pipeline is kept filled all the time, which is impossible. However, these techniques are appealing, in that they maintain the programmer's view of the system as essentially executing his program one step at a time, just as with the serial machine. Such machines can still be programmed in familiar languages, and thus exploit the large body of software which has been developed over the years.

A very different way to exploit parallelism is to abandon the concept of a single central processing unit which performs the computations. Instead, many processing units are used to perform portions of the job. The advances in technology which made hardware smaller, faster, and more reliable also made it cheaper. As this technology advances, it becomes feasible to consider building large numbers of processors, capable of working together to complete a computation [Uhr, 1984].

This approach allows much higher degrees of parallelism of computation, limited only by the constraints of the problem being solved. For many problems, this approach promises much faster solutions than those mentioned previously. However, in order to exploit this parallelism, new ways of solving problems, with the computations divided over many processors, are required. The means of dividing the problem over many processors is heavily dependent on the details of the computation to be performed. There is no easy or obvious extension to traditional computer architectures to be pursued in this area, as there is with the types of parallelism mentioned previously. Thus, significantly more work is required to exploit this type of parallelism. However, the potential payoff in terms of speed of computation is much larger, so this area is definitely worth investigating.

In designing large arrays of processors, many factors must be decided. Should all the processors be synchronized, executing the same instruction under the control of a central controller (SIMD machine), or should each operate independently, performing computations as required under its own control (MIMD machine)? Should data be stored in a single global memory, accessible by all the processors, in local memory available only to a single processor, or in some combination of these methods? How should the processors communicate? How much capability, and therefore complexity and cost, should be provided in each processor?

The answers to these and many other questions depend on the application for which the system is being designed. Computation and communication complexity vary dramatically for different types of problems. The features that make a system particularly effective in one application make it very ineffective for other applications. Efforts are underway to develop highly parallel systems which can be used effectively for a wide variety of problems, including PASM [Siegel et al., 1979], the NYU Ultracomputer [Gottleib et al., 1983], and CHiP [Snyder, 1982], among many others. However, even though these machines are general purpose, their benefits will still be restricted to the types of problems which can be solved by parallel arrays of processors.

There are many advantages in limiting a processing system to a particular application. It simplifies the design of the system, since the intelligence and flexibility needed to recognize and adapt to the requirements of different types of problems are not necessary. This can reduce the cost of the system significantly. Perhaps most significantly, it allows the design to be optimized for the requirements of a particular algorithm. The capabilities of each processor, the interconnections between processors, the control of the individual processors, the memory structure, and so on can all be tailored to the algorithm which the processor will execute. This requires that the designer select or devise an algorithm which is capable of solving the problem of interest with a high degree of parallelism, and design a system which can implement this algorithm efficiently. These two aspects of system design (algorithm selection and architecture design) tend to be interactive; both are current areas of research. A system designed as an efficient implementation of a highly parallel algorithm can give better performance than a general purpose system could provide for the same application.

The economics of a special purpose system must be studied carefully. A system capable of performing only a single function can be justified if either the user has enough work of that type to make good utilization of the system, or if the superior performance of the special purpose system is required for some compelling reason. An example of the first reason is image processing for LANDSAT data. NASA developed a very costly computer for just this one function, called the Massively Parallel Processor, or MPP [Potter, 1983]. The LANDSAT provides enough image data to keep this machine busy, so the fact that it can not be used for other purposes is irrelevant. An example of the second justification is the potential application of impedance imaging. If this can be developed into a useful technique for patient monitoring, it will be necessary to reconstruct the images quickly enough to detect and respond to changes in the condition of the patient. In such an application, the processor must be fast enough to respond to the real time situation, without regard to its adaptability to other functions.

In response to these types of requirements, many parallel algorithms and specialized architectures have been developed or proposed. The Parallel Element Processing Ensemble (PEPE) [Mariani and Henry, 1978; Vick and Cornell, 1978] was developed for tracking large numbers of objects detected by radar. In this system, the individual processors work independently, with little intercommunication, since the path of each object is independent of the others. Thus, the algorithms used are the same as for a single processor, with the addition of a scheme for keeping track of which processor is performing the computations for each object.

Many architectures have been proposed for image processing. This is a natural application of parallel processing, since digitized images contain large amounts of data, much of which can be processed in parallel. Examples of these systems are MPP as mentioned above, the Cellular Logic Image Processor (CLIP) series [Duff, 1978], and the Pyramid Machine [Tanimoto, 1983]. While these architectures are not directly applicable to my problem, the do offer valuable insights into what is required to get a large array of processors to operate effectively.

Some architectures have been devised specifically for the solution of matrix problems. This is another logical area for the application of parallelism, since matrices contain (potentially) large amounts of data. Systolic Architectures (actually a group of similar architectures) [Kung and Lieserson, 1980; Kung, 1982] use a two dimensional grid of very specialized processors to perform computations. Each processor receives operands from several ports, multiplies and adds them in predefined ways, and sends the result to predefined neighboring processors. The data of the problem is fed into the processing grid in such a way that the results appearing at the other side of the grid represent the solution to a problem of interest. This architecture has the advantage of very simple and regular communication requirements. This regularity is obtained from the assumption that the matrix in the problem is banded (a dense matrix is a special case of a banded matrix which can also be solved with this type of architecture). For systems which are extremely sparse even within the bands (including most finite element problems, as explained in Chapter 2), this architecture is less efficient, requiring a large number of processors and storage of many zeros in the band. It also requires that an entire row of the matrix be supplied to the processing grid for each time step, which could be difficult for large problems. Thus, systolic architectures are not well suited to the application in which I am interested.

Two systems have been proposed which directly address the solution of sparse linear systems resulting from finite element problems. The Finite Element Machine [Jordan, 1978a, 1978b] is a system designed to handle all aspects of finite element analysis. It uses 1024 general purpose processors, arranged in a two dimensional grid. Each processing element consists of a commercially available microprocessor (TMS 9900), with a floating point coprocessor (Am 9512), 32 KBytes of RAM, 16 KBytes of ROM, multiple I/O ports, and signalling circuitry [Adams and Crockett, 1984]. Each processor is connected directly to its eight nearest neighbors, and also to a global data bus for communication with more distant neighbors. Each processor performs the computations for a subset of the variables in the linear system. The architecture was originally intended to implement an iterative solution algorithm. Gannon [1980] suggested a direct solution method which could also be implemented on this system.

The second proposed system is called the Sparse Matrix Solver Machine [Amano et al., 1983]. This system also is designed for an iterative solution scheme. Its architecture is based on some general observations about the computations involved in this algorithm, particularly the data communications required. Processors communicate data by shared access to central memory. With many processors sharing access to memory, conflicts among several processors trying to access the memory present a potentially serious bottleneck to system performance. To overcome this, the authors propose a memory system which keeps multiple copies of data items, allowing several processors to access memory at the same time. By grouping the processors in clusters, and allocating more memory copies for data items within a cluster than for items from distant clusters, they claim to provide rapid communication without having to maintain an excessive number of copies of all data items.

In addition to these specialized systems, there has been a recent development in this area utilizing commercially available parallel processing hardware and an algorithm designed for implementation on that hardware. Pan and Reif [1985] devised an algorithm called Parallel Nested

Dissection for implementation on a system called the Connection Machine, a commercially available system providing 64,000 processing elements. They are currently in the process of implementing this system.

Another commercial design which could be applied to this problem is the Transputer, built by INMOS [INMOS Limited, 1984]. This is a general purpose processor with four bidirectional communications ports, all integrated on a single chip. The processor provides high performance, and the ports allow many processors to be connected in various ways to achieve highly parallel computation. This machine could be used as the basis for a parallel processor for the solution of large, sparse linear systems, by adapting one of the algorithms developed for this purpose to the capabilities of the hardware.

My Research

In examining these and other systems in detail, I found that each had limitations. Although each is a significant contribution to this field, none of them appears to provide the ultimate speed for the solution of large sparse linear systems. Some make use of general purpose processors, and so can not take advantage of all the parallelism available in the algorithm (see Chapters 2 and 3 for descriptions of the levels of parallelism available in a solution algorithm). Some are difficult to expand to maintain rapid performance as the size of the problem grows. Some require excessive amounts of resources for the functions accomplished. Details of these and other concerns with available architectures are presented in Chapter 6, in comparing these systems with the architecture I have proposed.

For these reasons, I determined that there was room for development of a new system for the parallel solution of large, sparse, symmetric positive definite systems of linear equations. This requires selection of a solution algorithm for this type of problem which would be applicable to a significant class of problems, and would allow the highest possible degree of parallelism in computations. Then it would require the design of an architecture which could implement this algorithm efficiently, and take advantage of the parallelism inherent in the computation to provide fast solutions to these problems. This combination of a highly parallel algorithm and an efficient hardware implementation of that algorithm should result in a system capable of solving such systems very rapidly.

A major concern of such a system is how the time for solution grows as the size of the problem increases. General direct solution methods, implemented on serial processors, require time which grows with the cube of the number of variables [George and Liu, 1981]. If this result held in Kim's application, and the size of the model grew to 10,800 nodes (e.g., to model the entire torso), the image reconstruction time would be more than one month. Better algorithms and more processors can combine to reduce this dramatically. But still, as the size of the problem grows, the size of the processing array must grow as well in order to maintain the best possible performance. Thus, the system must be designed for easy expandability.

My design is addressed to the need for the fastest possible solution to this type of problem. Some of the systems mentioned above are suitable for applications which can tolerate additional delays in computation in order to gain the benefits of lower cost or more generality of application. Rather than design another system with this same characteristic, I decided to design a system under the assumption that the combination of falling costs of VLSI components and the need for fast solution of some linear systems makes it feasible to dedicate a significant amount of computational resources to this problem. My design is intended for applications in which delays of minutes or hours for obtaining the solution to a large linear system are not tolerable. In particular, I assumed that it is possible for the size of the system to grow linearly as the size of the problem grows, and for the system to be optimized exclusively for this application, in order that it be capable of solving linear systems as quickly as possible.

My system is designed as a peripheral processor to a general purpose host computer. I assumed that this host machine would perform operations such as problem entry, permutations to the basic problem such as reordering the rows of the matrix and the corresponding elements of the vectors, and storing and displaying the results. Thus, my processor would be responsible only for performing the actual solution of the linear system of equations.

My research covers all aspects of the proposed system, including the algorithm to be used, the functions of the processors, the interconnections between them, and the detailed operation of each processor. In order to build the system, it would be necessary to design the processor in detail, down to the actual circuit level, build and test these processors, connect many of them together, interface them to a host processor, and use this system to run a variety of actual problems. These things are beyond the scope of my research project. This raises the problem of how to ascertain the performance of the proposed system.

Many systems are characterized in terms of the number of instructions or floating point operations they can execute per second. Unfortunately, these figures do not give a reliable idea of how quickly the system will solve a particular problem of interest. Furthermore, these measures can be exaggerated in many cases, since they result from measuring the time required for one processor to execute one instruction, extrapolating to a number of instructions per second, and multiplying by the total number of processors present. Thus, problems such as data dependencies, communication delays, and memory access conflicts are ignored.

To get a meaningful estimate of the performance of my proposed system, I decided to perform a detailed simulation of the system as it would operate to solve actual problems. The resource available for this simulation was the SIMULA programming language [Birtwistle et al., 1973] operating on a DEC-20 computer. This language is specifically designed for the simulation of discrete event systems, with many operations happening in parallel. It allows the modeling of the interactions between processes which have potential data dependencies, conflicts in access to resources, and so on. By carefully constructing a model of my system in this language, I could get a good idea of how it would behave and perform under actual operating conditions. With a few simple assumptions such as the speed of the system clock, it is possible to extrapolate measurements from this program to obtain the time the system would require for the solution of realistic problems.

Even with detailed simulation of the system, there is always the possibility that the design would not work as proposed once it was implemented in actual hardware. Problems of timing and synchronization of hardware are easily brushed aside in simulation. To avoid this possibility, I decided to construct a prototype of the processing element, using commercially available small and medium scale integrated circuits. This would help refine the design by uncovering deficiencies and finding solutions for them, and also serve to verify that the proposed system would actually work as intended.

Finally, there is the possibility that a well designed system will fail to give the promised performance in operation due to an unforseen bottleneck in such areas as loading data into the system or retrieving the results of the computation. Although I did not include these areas in the simulation, I do discuss them in detail.

The following chapters describe the selection of an algorithm for this system, the architecture I designed to implement this algorithm, the simulation and prototype verification of the capabilities of the system, and the issues involved in the operation of the system. Finally, they assess the potential capabilities of the system and compare this system with other possible systems which could be used in similar applications.

CHAPTER TWO - ALGORITHM

In order to achieve a fast solution to a problem by the use of many parallel processors, it is necessary to select an algorithm which allows a high degree of parallelism in the computations. Otherwise, many of the processors will be idle, so they will not contribute to the speed of the computation. Many algorithms have been developed for the solution of linear systems, with different properties which make them suitable for different applications. Each algorithm offers particular advantages and drawbacks which must be considered carefully. This chapter describes the algorithms I considered for my application, the reasons why I selected the one I did, and details of how the algorithm works. In order to justify the selection of this particular algorithm, I need to make reference to the properties of the type of problems I intend to solve with my system, so the chapter begins with a more detailed description of the problem.

Linear Systems from Finite Element Problems

The finite element method gives rise to linear systems with certain properties which can be exploited in their solution. The most important of these properties is that the matrix is sparse. A key aspect of the finite element method is that the equations governing the value of the field variable at a node involve only the values of the field variable at other nodes which are connected by a common element to that node. Consider a typical finite element model with triangular elements, as illustrated in Fig. 1 (in this figure, the circles represent nodes, and the lines serve as element boundaries). The equations for node 5 in this problem will involve the values of the field variable at nodes 1, 2, 3, 7, 8 and 9, since these nodes are connected to node 5, but not the values at nodes 4 and 6, since these nodes do not share a common element.

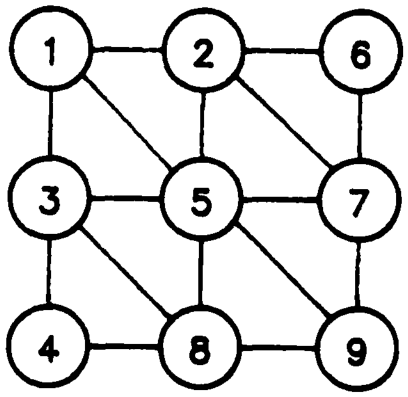


Figure 1 - Finite Element Model

In the small example of Fig. 1, this has little effect, but for a larger finite element grid, with the same structure but many more elements, the effect is substantial.

The matrix for the linear system has a row and a column for each variable in the system (i.e., for each node in the model). At each point in the matrix is a constant describing how the variables associated with that row and column affect each other. Since the variables do not affect each other at all unless the nodes are in a common element, most of the entries in the matrix will be zero. If the structure of Fig. 1 is expanded to produce a model with 10,000 nodes, each row will have no more than seven non-zero entries, so more than 99.93% of the entries in the matrix will be zeroes. This sparsity can and must be exploited in order to obtain an efficient solution.

A second significant property of the linear system comes from the fact that the model represents a physical system. In this case, the constant relating the field value at node b to that at node a is the same as the constant relating the values in the reverse order. If $a_{i,j}$ represents the value of the matrix entry for row i and column j, then $a_{i,j}$ will be the same as $a_{j,i}$; in other words, the matrix is symmetric. Also, in most applications, the finite element method gives rise to linear systems which are positive definite (matrix A is said to be positive definite if, for any vector x except the zero vector, $x^TAx > 0$) [George and Liu, 1981]. Matrices which are symmetric and positive definite have important properties for solution, which will be described below; they are referred to as SPD matrices.

Finally, the linear systems I am interested in are large. The range of currently available computers can solve systems of 1,000 to 100,000 or more nodes, depending on the time for solution considered tolerable. Systems with a few thousand nodes require times of a few minutes to a few hours on fast minicomputers such as the IBM 4381. Larger systems, involving tens of thousands of nodes, require times on the order of minutes on supercomputers such as the CRAY X/MP or large array processors such as the FPS-164/MAX. Duff et al. [1986] describe a problem involving approximately 360,000 variables, which requires roughly 40 minutes for solution on the FPS-164.

However, as described in Chapter 1, solution times of minutes or hours are not acceptable in some applications. In order to fit within the time and computational resources available, problems have been limited to coarse resolutions, and other compromises have been employed. In other cases, time-critical applications rely on a limited set of precomputed solutions to the problem for selected situations, in order to provide the rapid response needed. My architecture is intended to eliminate the need for such measures, so it must be capable of solving large systems quickly.

Algorithm Selection

The algorithms for solving linear systems can be divided into two general categories - direct solution methods and iterative methods. Each of these methods has advantages and disadvantages for particular applications, which must be considered in selecting a method.

Direct methods solve a system of linear equations by a series of operations which reduce the matrix to a form which can be solved in a straightforward manner. They require a fixed number of operations, depending only on the size of the problem, not the particular numerical values involved. Many specific methods of performing these operations, optimized for particular types of problems, have been devised. In particular, a good package of FORTRAN routines, called SPARSPAK [George and Liu, 1981], is available for the class of large, sparse, SPD problems which I am interested in. The advantages of direct methods include:

- They provide the solution to the problem in a fixed amount of time. Numerical instabilities may reduce the accuracy of the answer, but do not affect the execution time.

- They require a smaller number of computations than iterative methods for problems below a certain size. The dividing line, above which iterative methods can save computations, seems to be moving toward larger problems as more efficient ways of handling direct solutions are found.

- No special parameters, such as the relaxation parameters required in some iterative methods, are needed. Thus, once a system is working, it should work for all applicable problems without modifications.

- The process is performed in several distinct steps. For example, a series of operations may be applied to the matrix to reduce it to a product of two triangular matrices (LU Decomposition), then the linear system could be solved using these matrices (Back Substitution). For some repetitive problems, such as solving systems with the same matrix (A) but different right hand side vectors (b), the results of the matrix transformation steps can be saved, so only the solution steps need be rerun. In these cases, the solutions of the problems after the first one can be done very quickly.

Direct methods also have several disadvantages, including:

- In the process of forming the triangularized matrix, many of the entries which were initially zero become nonzero, due to "fill-in". The problem must be carefully arranged to minimize this, or the memory required to store the matrix can grow excessively. A problem with 1000 variables might have 5000 to 8000 nonzero entries, but if the entire matrix were filled in, there would be 1,000,000 nonzero terms. There are several techniques to minimize this fill-in [George and Liu, 1981], but it is still common for the filled in entries to outnumber the original nonzero terms. Storage is needed for these extra nonzeros. This phenomenon does not occur in iterative methods.

- It is difficult to find a high degree of parallelism in the solution steps used in these methods. Operations tend to work on at most the entries of one row or column at a time. Each step of the process depends on the results of the previous step. This means that, on average, only about 10 entries from the original matrix could be operated on simultaneously. This is not enough parallelism to achieve the speed I need for my system. There are only a few algorithms which achieve good parallelism; these are discussed below.

- The number of computations required for solution grows rapidly as the size of the problem increases. Golub and Van Loan [1983] show that for direct solution methods which do not exploit sparsity, the number of operations required grows as the cube of the number of variables in the equation (in other words, they require $O(N^3)$ operations). No matter how fast the single processor, this factor will limit the size of problems it can solve quickly.

- The finite wordlength of the computer can lead to roundoff and truncation errors in the computation. In direct solution methods, such errors tend to accumulate and cause inaccuracies in the final solution. This numerical instability requires higher precision arithmetic, which slows the computation down and increases the amount of storage required to solve the problem.

- Any change in the original matrix requires a complete re-solution of the problem. The solution for the previous problem provides no information which would speed up the solution to the revised problem. In the impedance imaging application, values in matrix A are revised for each iteration. If the changes in A are small, the changes in the solution to the linear system are likely to be small as well. However, direct solution methods can not take advantage of this fact to find the new solution more quickly.

- Direct methods require division, and sometimes square roots, in the computations. These operations take longer to perform than additions and multiplications, and also require more complex hardware or software.

I found only three direct solution algorithms which appeared to offer enough parallelism to be viable candidates for my system. The first of these is the use of LU decomposition in a systolic architecture. Kung and Lieserson [1980] show that this technique can solve a dense N x N linear system in time O(N), using N^2 processing elements. For a banded system, the time is the same, but the number of processors is related to the square of the bandwidth of the system (i.e., the distance between the main diagonal of the matrix and the farthest nonzero entry). This time is much faster than the $O(N^3)$ relationship which holds for a single processor.

However, in large problems, it may not always be possible to keep the matrix bandwidth small. For example, Fig. 2 shows two finite element problems, each with 25 nodes. In these figures, the lines between nodes represent connections between nodes (i.e., the graphs of the problems), rather than element boundaries. In both problems, the nodes have been numbered by the Cuthill-McKee algorithm [Cuthill and McKee, 1969], which is designed to

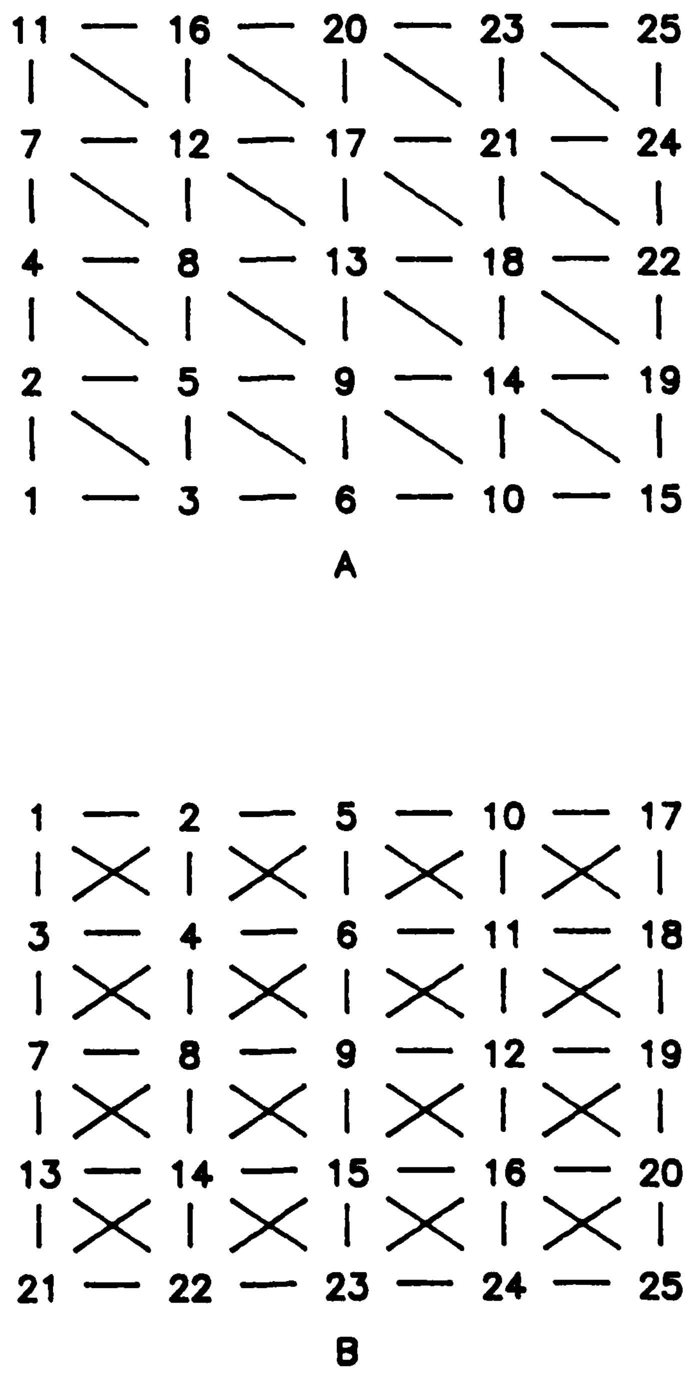


Figure 2 - Minimum Bandwidth Node Numbering

minimize the bandwidth of the matrix. The bandwidth of the resulting matrix can be observed by examining the graph of the problem and finding the largest difference between the node numbers of connected nodes. For the problem shown in Fig. 2a, the bandwidth is 5, so the systolic architecture would require 25 processors, the same as the number of variables in the system. But for the problem of Fig. 2b, the bandwidth is 9, so 81 processors would be required. This problem can cause the number of processors to grow much more rapidly than the size of the problem. Also, this architecture requires a memory system capable of supplying an entire row of the matrix to the array of processors in each time step. This memory bandwidth requirement would also rise as the size of the problem grew. Thus, this system would be very expensive to build, and perhaps more significant, very difficult to expand to meet the needs of larger problems. Therefore, I decided not to adopt this approach.

The second method I looked at is known as Nested Dissection [George and Liu, 1981]. In this method, the variables are numbered in such a way that the large problem can be divided into several smaller problems which are independent of each other. Several processors could solve these small problems independently, and then the results could be combined to form the overall solution. This method has two drawbacks for my application:

- It provides only a moderate amount of parallelism. For example, in a problem with 100 nodes arranged in a 10x10 square, only 16 independent 2x2 matrix problems are formed. If the nodes are arranged in other configurations than a square, there is even less parallelism in the initial stages of the solution.

- As the solution progresses, the nature of the computations needed changes from the parallel solution of a number of small, independent systems to the solution of smaller numbers of increasingly large, dense systems. This increases the complexity of the control of the processing and the communications between processors, and makes it difficult to keep all the processors busy performing useful work.

Gannon [1980] proposed an algorithm for highly parallel solution of a system of linear equations which is based on the above technique, and employs Givens Rotations. This method was proposed specifically for the application where many problems were to be solved with the same system matrix, but different right hand sides. In attempting to use this algorithm, I found that it was not particularly efficient for solution of a single problem. Also, it requires that the entire matrix be stored, even if the matrix is sparse. Since the problems I am considering are very sparse, this would cause a great deal of inefficiency.

Pan and Reif [1985] also proposed a system based on this algorithm, implemented on a commercially available parallel processor. A detailed description of this system, including a comparison with my proposed system, is presented in Chapter 6 of this thesis.

After investigating the above described algorithms, I decided not to employ a direct solution method for my application.

The other methods for the solution of linear systems are known as iterative methods. In general terms, these methods work by estimating the solution to the problem, and applying a series of refinements to this estimate on successive steps, until the estimate is judged by some criterion to be of acceptable accuracy. The methods differ primarily in how they determine what corrections to apply to each estimate.

Iterative methods can be further divided into two broad categories: stationary methods (e.g., Jacobi iteration, Gauss-Seidel iteration) and gradient methods (e.g., Method of Steepest Descent, Conjugate Gradient Method) [Jennings, 1977]. In addition, many techniques have been developed to speed up the rate at which these methods converge to a solution. The advantages of the iterative methods include:

- Only the original matrix must be stored. There is no fill-in from manipulation of the original matrix. Thus, much less memory is required to store intermediate results of computations.

- A high degree of parallelism is possible. At least for the stationary methods, the computation of the value of each variable for each iteration step is affected only by the values of the variables from other nodes in the same element. By assigning a processor to each variable, or a small group of variables representing neighboring nodes, the process of refining the estimated values of the node variables can be done in parallel. Thus, a large number of processors can be kept busy doing useful work using these methods.

- Each iteration of the process produces a closer approximation of the final solution to the problem. In some instances, it might be sufficient to produce a rough estimate of the solution, which could be done in just a few iterations. For the impedance imaging application, each refinement in the element impedances is likely to have only a small effect on the solution to the finite element problem. In this case, the previous solution could be used as a starting point for the next problem, and the answer should be obtained in a small number of iterations. Only iterative methods can take advantage of these situations to save computation time.

- As the size of the problem grows, the total number of computations required grows more slowly for iterative methods than for direct methods [Jennings, 1977]. Thus, for the large problems I am interested in, iterative methods require fewer computations.

- Roundoff and truncation errors due to the finite wordlength of the computer tend not to accumulate [Westlake, 1968]. In effect, these errors are the same as differences between the initial guess and the final solution, so successive iterations tend to eliminate them, rather than magnify them. This can allow the system to use shorter wordlengths, thus increasing the speed of the computations and reducing the amount of memory required. (Some researchers object to stating this empirical result as a generality; cf. Wilkinson [1973].)

- The computations involve only multiplication and addition, which are faster and cheaper to implement in hardware than division and square roots.

These factors make iterative methods attractive for my application. However, there are some disadvantages to these methods which must be considered:

- Many of them depend on careful selection of parameters to obtain efficient operation, such as a weighting coefficient to help apply the optimum correction at each iteration step (e.g., the relaxation parameter for successive over-relaxation). Poor choice of such parameters can cause the convergence to be slow, or the problem to fail to converge at all. The choice of these parameters depends on the particular problem being solved. If the problems for which the system is needed differ enough to require widely varying parameters, then the choice of this parameter could present a significant obstacle to attaining good performance from the system.

- It is difficult to know when to stop the iteration. It is not possible to know absolutely that the exact solution has been obtained. Therefore, the algorithm needs to keep track of some measure of the error in the solution, or the amount of change which occurs in the solution at each step. These measures can be deceptive in some cases. For example, if the system is converging very slowly, there might be little change at a step, and the algorithm might incorrectly decide that the solution has been reached, when in fact it is far away. Another possibility is that the algorithm could oscillate about the correct solution, not making any progress toward it; then the iteration would go on indefinitely, even though it was as close to the correct solution as it could get.

- The accuracy of the computed solution depends on using the correct number of iterations. Due to the problems mentioned above, the solution might contain substantial inaccuracies. Thus, knowing when to stop iterating is important both for the speed of the computation and the accuracy of the results.

- If several problems must be solved, involving the same matrix (A) but different right hand side vectors (b), iterative methods can not take advantage of the solution of one problem to speed up the solution of successive ones.

Despite these limitations, I determined that iterative methods were the most promising for my application. Comparing the properties of direct and iterative methods for repetitive problems, it can be seen that direct methods are more efficient when successive problems involve the same matrix but different right hand sides, while iterative methods are more efficient for multiple problems with small changes in the matrix and the same right hand side. The latter is the case for impedance imaging, since the values of the element impedances are updated for each step, resulting in changes in the matrix of the linear system. Since these problems provided the original motivation for this research, it is appropriate to select the solution algorithm which best fits this application.

Many sophisticated algorithms have been developed in recent years based on these fundamental techniques [Hageman and Young, 1981;Golub and Van Loan, 1983]. However, most of these algorithms have been designed specifically to reduce the total number of arithmetic operations needed, since this is the only way to improve execution speed on a single processor. Such refinements often make the algorithms unsuitable for a parallel architecture, either by reducing the amount of parallelism possible, or by increasing the amount of intercommunication required between processors. For example, the gradient methods calculate an optimum search direction for each iteration step, in order to minimize the number of iterations required. This extra calculation requires the computation of an inner product, based on the last iterated value of the solution vector. However, if the N variables of the solution are distributed over N processors, all the variables must be transmitted to one processor to compute the inner product, which requires $O(N\log_2 N)$ steps [Kung et al., 1985]. Thus, the time required for each iteration would grow faster than the number of variables in the system. This factor makes methods such as the Conjugate Gradient method unsuitable for implementation on a parallel processor.

In a parallel architecture, the algorithm must be evaluated not based solely on the total number of computations required, but also on the amount of parallelism allowed. Based, on these considerations, I eventually limited the candidate algorithms to two straightforward ones, Jacobi iteration and Gauss-Seidel iteration with Successive Overrelaxation.

Jacobi iteration is the simplest iterative method, and allows a great deal of parallelism. The value of the next iteration for each variable is a linear combination of its previous value, the corresponding entry from the right hand side of the equation, and the values of variables for neighboring nodes, as follows:

$$x_i^{k+1} = b_i - \sum_{j \neq i} a_{i,j} x_j^k \qquad (2)$$

Thus, all nodes can be updated simultaneously, send their results to the neighbors which need them, and then repeat this process. Unfortunately, the convergence properties of this algorithm are poor (see results below).

Gauss-Seidel iteration is very similar to Jacobi iteration. The basic computation is the same, except that the nodes are updated in a particular order, and the most recent value of each neighboring node variable is used in each computation. For example, if the computation for node

5 involves the values of variables for nodes 3 and 7, the newly updated value for node 3 and the old value for node 7 will be used, since these are the latest values available. This technique speeds up to convergence somewhat, and is a very natural technique to use with a single processor (or, as it was originally implemented, by manual computation), which must update the nodes sequentially anyway.

The method of successive over-relaxation (SOR) modifies this algorithm by scaling the corrections to be applied at each iteration by a "relaxation parameter". A properly chosen parameter can speed the convergence of the iteration dramatically. Jennings [1977, page 194] gives an example of a type of problem for which Jacobi iteration, Gauss-Seidel iteration, and SOR require 7000, 3500, and 150 iterations respectively. In addition to being faster, this method has the advantage of being more reliable. Jennings shows that the method will always converge for an SPD linear system, provided the relaxation parameter chosen is within the range $0 < w < 2$.

This advantage in convergence rate makes SOR a very attractive candidate algorithm. The concern then becomes the lack of parallelism in the method, due to the requirement to update the variables in order. If it were necessary to update every node before any node could again be updated, this would be a prohibitive limitation. This would be the case if the matrix were dense, i.e., if the values of the variable at all nodes affected the values at all other nodes. This is not the case in the problems I am studying. Since the matrix is sparse, the value of the variable at a node is related to the value at only a few other nodes. When this small number of nodes has completed an iteration, it may begin the next iteration. For example, consider the arrangement of nodes if Fig. 1, and assume that all nodes have completed the same number of iterations. Node 1 must be updated before nodes 2 and 3, since it is connected to both of them and has a lower node number. When node 1 has been updated, nodes 2 and 3 can be updated, followed by nodes 4, 5 and 6. At this point, node 1 has all the data it needs to complete its next iteration, without waiting for nodes 7 through 9 to be updated.

I wrote a small simulation program in BASIC for the IBM PC computer to illustrate this process in action for various interconnection patterns and node numbering schemes. One version of this program (Appendix A) used 36 nodes in a pattern similar to Fig. 1, numbering along diagonals of the grid of nodes which were perpendicular to the diagonals of the triangular elements (for convenience, I refer to this as "diagonal ordering" in the rest of this thesis). In this scheme, there is a significant delay between the start of the iteration and the first update to the node in the lower right corner, but then subsequent iterations proceed in "fronts" through the grid, with one third of the nodes updated each time step. Another possible ordering would start in the upper right corner, and follow along diagonals parallel to the diagonals of the elements. This proved to be a very poor ordering, allowing only one or two nodes to be updated at each step.

A different type of ordering achieves the same degree of parallelism as that used in the Appendix A program, but avoids the delay in the update of the last nodes. This ordering is based on a node "coloring" scheme, which is designed to minimize the dependence between neighboring nodes [Adams and Ortega, 1982]. The nodes in the problem are marked with different colors in such a way that no two nodes in a given element are the same color. Then, all the nodes of a given color are numbered before the nodes of the next color. The number of colors required to do this depends on the connectivity between nodes. In the simplest case, with each node connected to only its four nearest neighbors, two colors are required, and terms such as "two-color", "red-black", and "chessboard" [Jennings, 1977] ordering are used. For more complex interconnections, three or more colors are required, and the resulting ordering is called a "multi-color" ordering.

Figure 3 illustrates this process for a model similar to that in Fig. 1. R, G, and B represent the colors red, green, and black. In Fig. 3a, the nodes have been colored

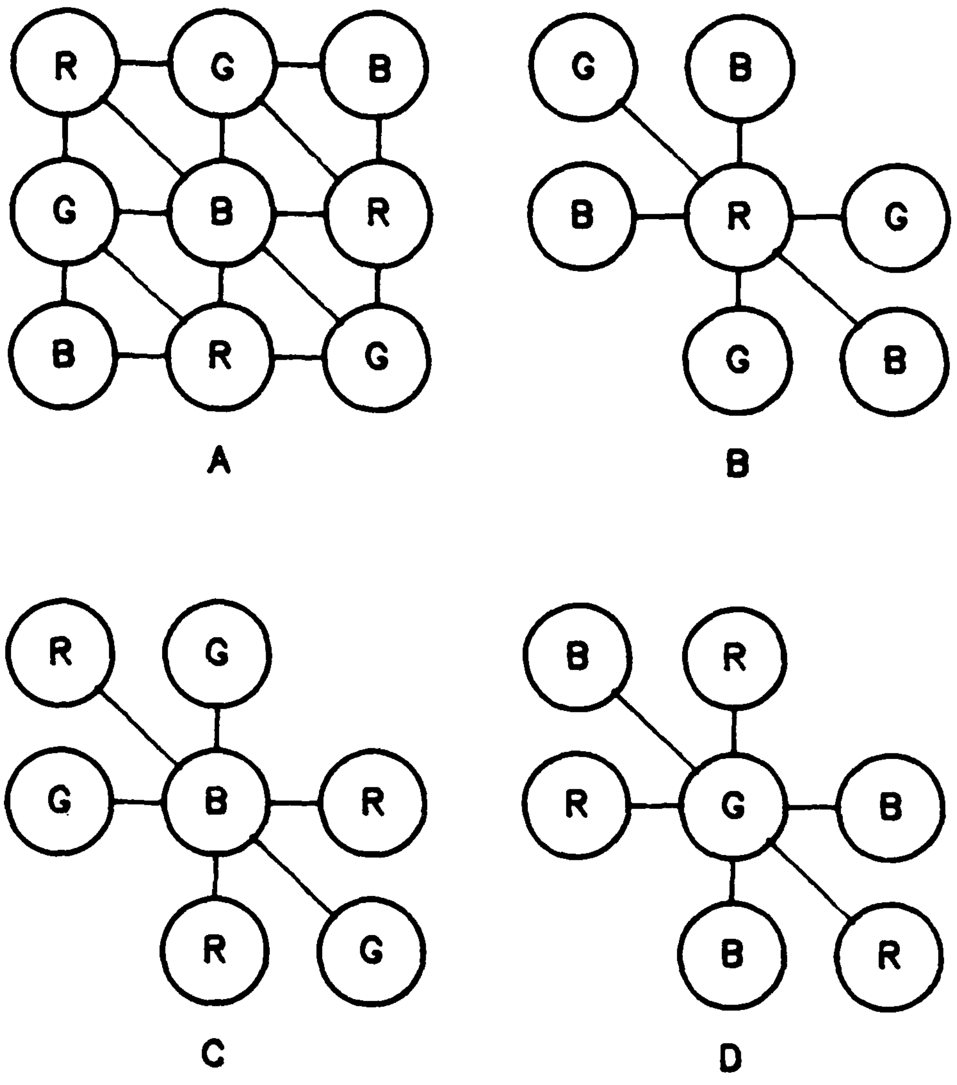


Figure 3 - Node Color Ordering

in such a way that each element has only one node of each color. Using this coloring scheme, all the red nodes could be numbered consecutively, followed by all the black nodes, and finally all the green nodes. In this way, each color of node is connected to, and hence affected by, only nodes of the other two colors, as shown if Figures 3b - 3d. Now asssume that all the nodes in the problem have completed the same number of iterations. Since the red nodes have the lowest numbers, they must be updated first, and require the previous iterated value of the variables for the black and green nodes, which are already available. Thus, all the red nodes can be updated on the next step. The black nodes have the middle range numbers, and require the new values for the red nodes, and the old values for the green ones. Once the red nodes have been updated, all the black ones can be updated on the next step. Finally, the green nodes can be updated, since the needed values from the red and black nodes are available.

The program in Appendix B illustrates the operation of the SOR algorithm with this numbering scheme. As described above, it allows updating of one third of the nodes on each step, but this happens from the beginning of the iteration, with no delay for the propagation of a "front" through the network. Thus, this method allows I iterations to be completed in 3I steps. Furthermore, after each 3 steps, all the nodes have completed the same number of iterations, unlike the case with the ordering shown in Fig. 1. This could serve to simplify the determination of when the iteration has converged.

Thus, even though the SOR method does not achieve the level of parallelism of the Jacobi iteration, for the type of problem of interest the difference is small. Schendel [1984] presents parallel versions of these algorithms, and tests them on a model problem which he devised. He shows that for this type of problem, SOR is clearly superior to Jacobi iteration in terms of execution speed when adapted to parallel processors.

As mentioned above, the two basic problems with iterative methods are the choice of the parameters (in this case, the relaxation parameter), and the determination of convergence. The SOR method is widely used for solving linear systems on general purpose, single processor computers. In this environment, users have developed many ways of coping with these issues. Unfortunately, many of these techniques assume that a single processor can access any element of the solution vector with equal speed. If the solution is distributed among many processors, this is no longer the case. The necessity to contend with this situation must be considered.

The choice of the optimal relaxation parameter depends on the problem to be solved. If the eigenvalues of the matrix are known, this parameter can be computed simply as:

$$w = 2 / (1 + (L_1 L_n)^{1/2}) \tag{3}$$

where L_1 is the smallest eigenvalue in the system, and L_n is the largest. This equation is dominated by the value of the smallest eigenvalue. As this value approaches zero (it is known to be positive; cf. Jennings), w approaches 2. Unfortunately, in many cases, the eigenvalues are not known ahead of time, and their computation can take longer than the solution of the linear system [Jennings, 1977], so this approach can not be used.

Carre [1961] describes a method for computing the optimum relaxation factor dynamically as the iteration progresses. This method involves the computation of matrix norms for the differences between successive iterated values of the solution to the problem. As with the computation of the optimum search direction for the conjugate gradient method, this requires excessive communication on a parallel processor. Also, this method only works for problems which can be ordered with a two color ordering, which is not true for most finite element problems. Thus, this method can not be used for the problems my system is designed to solve.

However, many users of the SOR method have determined empirically good values of the relaxation parameter for their applications [Kim, 1982; Kim et al., 1986]. In these cases, the known value could be used equally well in a parallel computer. If the optimum value of w is not used, the primary effect is to require more iterations for convergence. This could be offset significantly by the speed of the parallel processor in performing the iterations in the first place. Thus, the need to select a good relaxation parameter does not prohibit the use of SOR in a parallel architecture.

Knowing when to stop the iteration process can be a difficult problem. Ideally, when solving the problem

$$Ax = b$$

we would like to compute the error as $||b - Ax^k||$, where x^k is the k-th iteration of the vector x, and $||\cdot||$ represents some vector norm (cf. [Stewart, 1973]). However, this computation is very slow. Instead, convergence is usually determined by observing the progress of $||x^{k+1} - x^k||$. Use of the "infinity norm" [Stewart, 1973] allows this test to be applied quickly. Saltz et al. [1986] have devised a method which allows this type of test to be implemented in a distributed architecture with minimal delay. I examined this proposed method, and believe that it could be usable for my application. The details of this algorithm, and its application to my system, are described in detail in Chapter 6 of this thesis.

Based on the above considerations, I selected the SOR algorithm for use in my design. It provides an excellent combination of total computations required, parallelism allowed, convergence properties, and adaptability to different problems.

Details of the SOR Computation

The basic computation to be carried out to update one variable under the SOR method is given by the following equation:

$$x_i^{k+1} = (1-w)x_i^k + wb_i - w\sum_{j=1}^{i-1} a_{i,j}x_j^{k+1} - w\sum_{j=i+1}^{N} a_{i,j}x_j^k \qquad (4)$$

where w is the relaxation parameter, subscripts refer to locations within vectors or matrices, and superscripts refer to iteration numbers. Careful examination of this equation reveals certain aspects which can be used to enhance the computation speed in the case where this algorithm is distributed so that one processor handles the computations for each element of vector x.

One of these is that the computation consists of a series of additions and multiplications. Clearly, these can be executed in any order without altering the final result.

As soon as the value x_j is received, it can be multiplied by the appropriate matrix entry, and the product added to a partial sum. When all the required terms have been added to this sum, it can be multiplied by the relaxation parameter, and the calculation completed. Thus, the processor assigned to the computations for a variable need not be idle while it waits for data needed in its computation. Instead, it can use this time for computing the first two terms in the equation, comparing its latest iterated valed to previous values to determine whether the process has converged, and so on.

Since the matrix of the system is symmetric, if the value of x_j is needed for the computation of x_i, then the value of x_i will also be needed at node j. Thus, it is possible at the beginning of the computation for each processor to know (a) what data it needs to perform its computations, and (b) where its results need to be sent. This suggests that the processors can be designed to handle data communication efficiently - processors need not request the data they need, but rather, can send the data automatically to all the processors which need it.

Summary and Conclusion

This chapter has presented the possible solution algorithms which could be employed in a parallel processor for the solution of large, sparse, linear systems. Based on the characteristics of the particular type of problem for which this system is being designed, the successive overrelaxation algorithm is selected. The combination of

- a high degree of parallelism,
- the ability to keep processors busy with the intermediate portions of the calculation,
- good convergence and numerical properties,
- the absence of fill-in in the sparse matrix,
- the ability to exploit the results of previous solutions when the values in the matrix have changed slightly,
- the relatively small number of computations required for large problems, and
- the predetermined communication patterns

make the SOR algorithm attractive for this application.

In order to take advantage of these aspects of the SOR algorithm, an architecture is required which can implement the algorithm efficiently. The following chapters describe an architecture devised specifically for this purpose.

CHAPTER THREE - ARCHITECTURE

In this chapter, I present the architecture which I have devised to:

- execute the SOR iterative method to solve large, sparse, SPD systems of linear equations,
- allocate one processor to each variable in the system,
- achieve maximum parallelism of computation,
- minimize the overhead associated with the computation, and
- utilize a processor which can be built on a single VLSI chip.

Architecture Considerations

One of the fundamental concepts to be incorporated in my design is that each processor should be implemented on a single VLSI chip. This is important for considerations of both cost and speed. If a processor is divided across two or more chips, the need for communication and synchronization between the chips can slow down the operation significantly [Hennessy, 1984]. Clearly, the use of more chips per processor increases the cost of the system, due not only to the cost of the chips themselves, but also to the cost of support such as circuit board space, packaging, power consumption, and so on. For an architecture which depends on a high degree of parallelism, the cost of each processor should be held to a minimum. VLSI technology has advanced to the point where sophisticated processors can be built on a single chip, so one chip per processor appears to be the optimum way to implement a highly parallel architecture.

This is not meant to imply that there can not be more than one processor for each chip. As the scale of integration improves, it could be possible to include several processors on a single chip. Alternatively, the use of wafer scale integration would allow several chips to be included in a single package, thus incorporating several processors. This technique can best be employed if the individual chips on the wafer can be interconnected to the maximum degree possible. A chip designed to be connected directly to other identical chips would be the ideal candidate for this level of integration.

In the reports of other researchers in the field of parallel architectures, one almost universal observation is that the architecture must pay careful attention to the requirement for intercommunication among the processors. If the provisions for this communication are not adequate, this can present a bottleneck which severely limits the performance of the machine. For example, a model of the performance of the Finite Element Machine indicated that as the number of variables assigned to a processor was reduced, the execution time could actually increase due to the time required for communication [Adams and Crockett, 1984].

From the description of the finite element method and the sample problems shown in Chapter 2, it can be seen that the communication requirements for this algorithm tend to be between nodes which are close together in the grid. All nodes are connected to some of their nearest neighbors; some are connected to more distant neighbors as well. This suggests that the processing elements themselves be arranged in a grid, since this would place the processors which need to communicate most frequently close together. For two-dimensional finite element grids, a rectangular two-dimensional grid of processors makes it possible to assign variables to processors in a way which preserves this locality of communication. This is the arrangement of PEs which I selected for my architecture.

The intent of this arrangement is to allow a direct mapping of the nodes in a finite element grid to the processing elements. With the PEs arranged in a grid, it is natural to refer to PEs as nodes in the processor array. Each element in the vector which is the solution to the linear system represents the value of the field variable at one node in the finite element grid. The computations for each node value are assigned to one node in the network of processors. Thus, in the discussions which follow, the word "node" will be used to refer to either a point in the finite element grid, a variable in the linear system, or a processor in the network. In cases where it is important to distinguish between these meanings, I will be specific about the terminology used.

Given the arrangement of processors in a two dimensional grid, the question to be considered becomes what sort of communication paths to provide for these processors. There are some tradeoffs to consider in this area. Clearly, the data paths should be matched to the requirements of the algorithm. The scheme chosen must be flexible enough to handle all the required communication. However, if some data paths are used much more frequently than others, the design should be optimized so that the performance of the frequently used paths is not degraded by the overhead needed to handle the rarely used ones. Many interconnection schemes have been devised for various applications [cf. Uhr, 1984; Siegel, 1979; Feng, 1981]. The problem is in choosing the best one for this application.

Some constraints are placed on this by the need to implement each processor on a single chip. This primarily limits the number of pins available for input and output of data, and the number of independent I/O ports which can be controlled. If each processor has many ports, it can dedicate only a few I/O pins to each. Furthermore, only a small amount of die area can be dedicated to the control of each port, so the ports will not be capable of performing complex functions independently. A large number of ports can allow the processor to be connected to many others with which it must communicate, but the communication with each will be slowed by the unsophisticated control and the small number of pins. Restricting the data paths to a small number allows more sophisticated processing at each port, and simplifies the layout of the system, but may cause excessive delays in routing messages over longer distances. These tradeoffs in the design of the communication paths between processors must be weighed carefully, and the system must be designed to minimize the drawbacks inherent in the chosen approach.

One significant source of delay in many architectures is the fact that the same processor which performs the computations essential to the algorithm also performs control functions for interprocessor communication. Thus, whenever data must be sent or received, the processor must stop performing computations to service the I/O request. Some computers avoid this delay by using special purpose communication processors to handle data communications so that the CPU can concentrate on data manipulation. This concept of providing special purpose functional units, capable of performing communication functions without intervention from the CPU, could minimize the impact of data communication requirements on the performance of a parallel architecture if implemented efficiently.

Another factor to consider in the basic selection of an architecture is the choice between so-called SIMD and MIMD architectures [cf. Hwang and Briggs, 1984 or Uhr, 1984]. In the SIMD (Single Instruction stream, Multiple Data stream) approach, all the processors execute the same instruction under the control of a single central controller. The primary advantage of this technique is that it keeps each processor very simple. There are several disadvantages. For more complex computations with data dependencies, it is often necessary to shut off many of the processors on each step while they wait for results from their neighbors, thus reducing the amount of parallelism achieved. The need to supply data to many processors at the same time requires very wide bandwidth memory. The communication paths between such processors are very limited, since the processors do not have the ability to utilize longer paths, so communication can be slow.

MIMD (Multiple Instruction stream, Multiple Data stream) designs allow all the processors to execute their own programs, depending on each other only for data. This allows much more flexibility in the utilization of each processor. If the processors interact only slightly, the architecture is described as "loosely coupled" - such systems are not applicable to my problem, since the processors do not cooperate on the solution to a single large problem. If the processors do interact, the system is "tightly coupled". The problems associated with this approach are that the control of the processors is much more difficult, the time required to download programs to a large number of processors can be prohibitive, and there is much higher potential for conflicts between the processors in attempting to access resources such as shared memory. The individual processors are more complex and expensive than those used in SIMD machines. The systems built to date mitigate these problems by keeping the number of processing elements small, e.g., the C.mmp (16 processors), Cray X-MP (2 or 4), IBM 308x (2 or 4) and Denelcor HEP (16) [Hwang and Briggs, 1985].

The ParSOR Architecture

Based on the above considerations, and the details of the SOR algorithm as described in Chapter 2, I have designed an architecture for the solution of large, sparse, symmetric positive definite systems of linear equations. I consider this to be the optimum architecture for the fastest possible solution of the type of problem described in Chapter 2. For convenience, I refer to this architecture as the ParSOR (for parallel SOR) architecture. The basic features of the architecture can be summarized as follows:

- A separate processing element (PE) is assigned to the task of performing the computations for each variable in the system.

- Coefficients related to the computations for each processing element are stored in that PE (i.e., the nonzero matrix entries for the appropriate row of the matrix, the initial value of the node variable, the element of the right hand side vector, etc.)

- The PEs are arranged in a two dimensional rectangular grid (Fig. 4). Each PE is directly connected to its four nearest neighbors (up, down, left, and right).

- Each PE is partitioned into two units - an Arithmetic Unit (AU) to perform the multiplications and additions required by the SOR algorithm, and a Communication Unit (CU) to transfer the data between PEs as required to complete the computations (Fig. 5).

- The AU performs computations as data becomes available (i.e., as data items are received from the CU). It has no global memory either for data or program storage. Its program is built in, so no program loading is required.

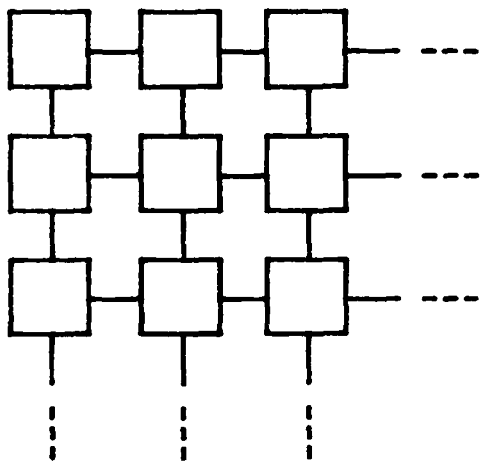


Figure 4 - PE Interconnections

- The CU performs all communication functions without intervention from the AU. Thus, the AU can be busy performing computations whenever it has data available.

- The CU has separate communication processors (CPs) to interface with each of its four directly connected neighbors. Each CP is capable of performing its functions independently of the others. For example, all four CPs can send messages to their respective neighbors simultaneously.

- Each CP can communicate with the AU, to send it data which it needs and receive results from it to be sent to other PEs. CPs can also communicate with each other to route messages which must be sent to more distant

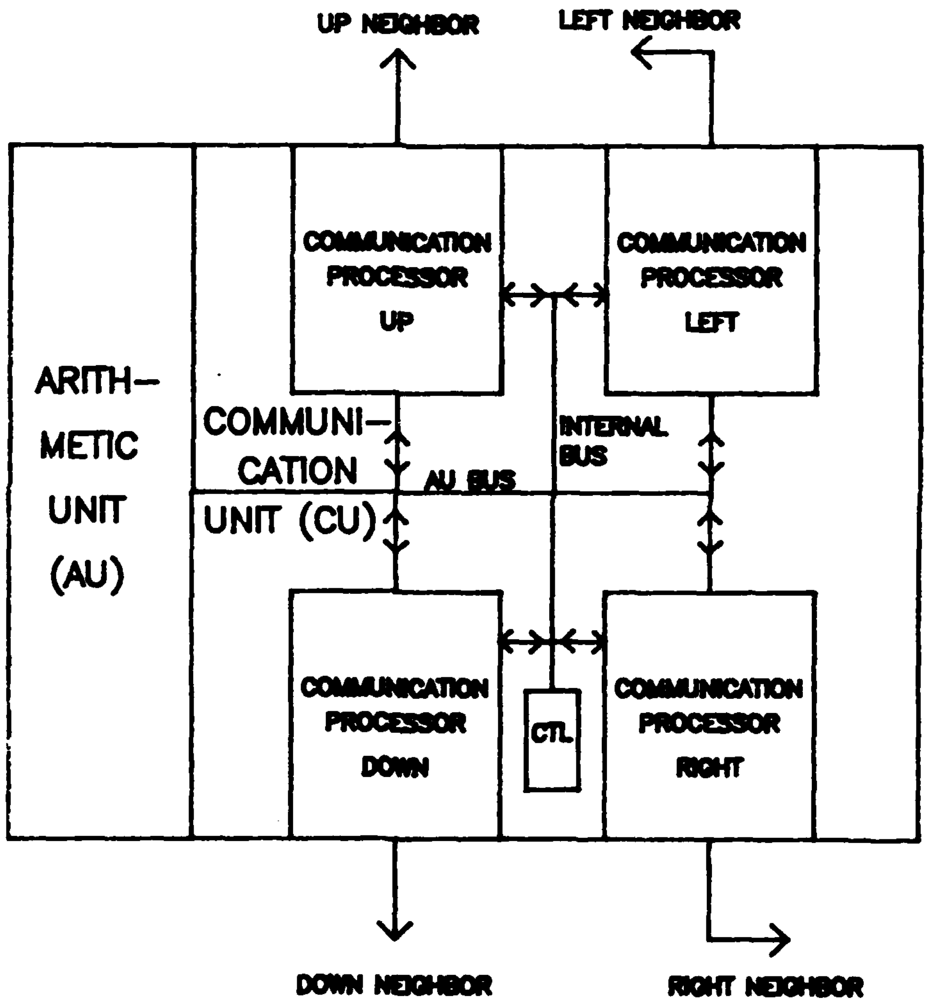


Figure 5 - Processing Element

locations than the nearest neighbor in the network.

- Messages sent through the network contain values of the variables being computed, identifiers to indicate which variable they represent the value for (i.e., their subscript), and routing information to indicate where in the grid of processors they should be sent. The CPs are capable of routing all messages correctly based on this data, with minimal delays.

- Messages contain a small number of control bits, indicating whether the message is a part of the iteration calculation, or a data item to be loaded in the array or returned to the host. The CPs respond to these bits to perform the correct processing of all messages.

The following sections describe the details of this architecture.

Arithmetic Unit

The basic function of this unit is to perform the computations for updating one variable for each iteration. As shown by equation 3 (chapter 2), the computation consists of a series of multiplications and additions. There are several terms which are computed in the iteration:

$(1-w) \times x_i$, $w \times b_i$, and the summations of $w \times a_{i,j} \times x_j$ (I use subscript i to refer to the variable computed by the AU, and subscript j to refer to variables computed at other nodes). A few things can be done to simplify this computation. The quantity (1-w) can be stored to avoid one subtraction. The term $w \times b_i$ does not change as the computation progresses, so it can be computed once and stored. Finally, since w is multiplied by all the terms in the summation, this multiplication can be saved until the summation is complete and then performed at the end.

The CPs are responsible for supplying the values of $a_{i,j}$ and x_j to the AU. Thus, the AU can wait for a flag indicating that an operand pair is available, read in the values from the CP, and start the multiplication. The AU must know how many such data pairs to expect for each iteration, so that it can tell when an iteration is finished. As it receives data items, it keeps track of how many of the pairs it has multiplied and entered in its summation. Since several CPs supply data to each AU, the AU needs a small buffer to store operands received while it is performing a multiplication. When all the terms have been incorporated in an iteration, the AU signals all four CPs that it has an answer ready for them. It then sends this answer to the CPs, and is finished with an iteration.

Once the AU finishes an iteration, it must wait for data from its neighbors to start the next iteration. It can use some of this time to perform the multiplication of $(1-w) \times x_i$. It can also use this time to compare its most recently computed value with the value it computed on the previous iteration. This is necessary for determining when the iteration has converged (see Chapter 6). As soon as any data arrives from a neighboring node, the AU can continue computing. Thus, the arithmetic unit at each node can stay busy a substantial portion of the time.

There is basically nothing novel in the functioning of the AU. It consists of a multiplier, adder, a small amount of storage, and some logic to implement processing of the data received, keep track of its status, and interface with the CPs. Thus, I have not given a great deal of consideration to the details of the design of this unit. The only detail which could vary substantially in the implementation of this unit is the capacity of the multiplier. Many different multiplier designs are available, which offer tradeoffs between speed and complexity (i.e., chip area). Which of these is implemented depends on how much area is available to the chip designer, and how much complexity he chooses to put into the chip. I have considered several levels of performance possible from different multiplier designs in evaluating this architecture. Other than this, I have not studied the design details of the AU.

Communications Unit

The Communications Unit handles all the communication of data between PEs. Since each PE communicates with its four nearest neighbors, the CU is divided into four Communications Processors, or CPs, as shown in Fig. 5. Each CP handles the direct communication with the corresponding CP of one of the neighbor PEs. Each is designated by the direction in which the neighbor with which it communicates lies - up, down, left or right. The CPs also communicate with the other CPs in the same PE over a shared data bus. Another data bus is used to provide the communication between the CPs and the AU.

When the computations for one iteration of a variable are finished, the result is sent to the nodes which need that value. The result consists of the value of x_i. In order to use this value, the receiving nodes need to know which value it is, i.e., the subscript i. Therefore, this value is appended to the message. If the value must be sent to a distant node, some means of specifying the destination is needed. Since the destinations can be precomputed, I implemented this simply by including a number of rows (R) and a number of columns (C) of displacement in the network which the message should travel. Thus, a complete message would contain the following information (the sizes of these data items are discussed later in this chapter):

x_j	j	R	C

For a message arriving at a processing node, two separate activities are required. The value of the index j must be examined to determine whether the value x_j is needed for the arithmetic computation at that node. Also, the values of R and C must be examined to determine whether the message should be sent on to other nodes, and updated to reflect the fact that the message has moved one step closer to its ultimate destination. These two determinations are independent of each other, and they should not require the intervention of the processor performing the iteration computations (i.e., the Arithmetic Unit).

Communications Processor

The CP is designed to have the capability to perform all of these determinations, and route the data as needed. It is provided with several functional units which handle the various aspects of these tasks, as shown in Fig. 6. The following paragraphs describe the operation of these units.

External Bus Interface - This unit provides the communication with another processing element. A single set of data lines is used for communication in both directions between the two processors. Therefore, some means of access

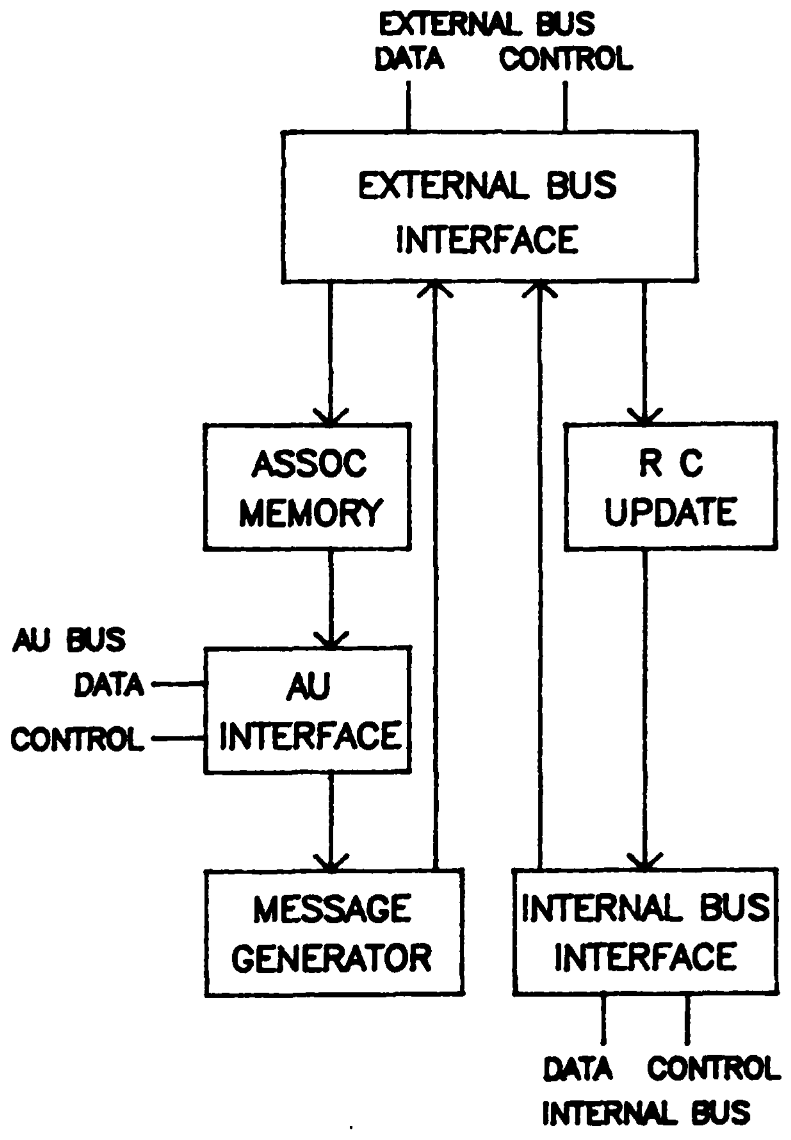


Figure 6 - CP Functional Units

control is required. This is implemented by designating some CPs as bus masters, and the others as slaves. Control lines denoted "Request", "Grant", "Data Ready", and "Buffer Full" are used to allow the processors to coordinate their use of the data lines efficiently and prevent conflicts. When the master wishes to send a message, it check the status of the Buffer Full flag, which indicates that the slave has no place to store an incoming message. If the slave has set this flag, the master must wait until it is cleared before sending data. Once this flag is clear, the master places the data on the data lines, then asserts the data ready signal, causing the data to be latched in by the slave. If there are fewer data pins than bits in the message, this last step is repeated until the entire message has been sent. Succeeding portions of the message can be sent at a rate of one each clock cycle, since there is no further need to gain access to the bus, until the entire message has been transmitted. When the slave has a message to send to the master, it first asserts a request for the bus. When the master is ready to receive, it sets a grant signal. Then the slave sends the message in the same way that the master does. This coordination is required to prevent both units from attempting to drive bidirectional data or signal lines at the same time. The CPs for communicating up and left are designated as slaves, and those for down and right as masters.

In order to reduce the number of input/output pins for a VLSI implementation of this architecture, the data lines can be multiplexed. For example, a 60-bit message can be transmitted over 20 lines in three cycles. I did not limit the design to a single bus width, but modeled the performance as the number of bus cycles needed to send a message varied. Overhead tasks, such as gaining access to the bus, need not be repeated for each cycle, so this does not cause an excessive delay. Data received by this unit is stored in a register for use by the Associative Memory and the RC Update unit, and released when they have both signalled completion of their tasks.

Both the Message Generator and the Internal Bus Interface (see below) produce messages which must be sent out from the external bus interface unit to neighboring PEs. When one of these message producers has a message to send, it signals this fact to the external bus interface, which then moves the message to its output buffer. As soon as the data is stored in the output buffer, the unit which originated the message can continue its processing. Then the external bus interface initiates the process of sending the message to the corresponding CP in the neighbor PE.

Associative Memory - This unit examines the value of the index, j, of any message which arrives in the input buffer of the external bus interface unit, in order to determine whether the data is needed by the AU. It does this through the use of a small "associative" or "content addressable" memory. If the value of j, as received by the external bus interface, is stored in this memory, it returns an index to a small conventional memory. This index gives the location at which $a_{i,j}$ is stored. Since the memory is associative, no searching is required, and the index, along with a signal indicating that the index was found, appear in one clock cycle after the data arrives in the register. In one more clock cycle, the value of $a_{i,j}$ is retrieved from the conventional memory. The associative memory unit then forwards the data to the AU Interface and singals that unit that the data is needed by the AU.

RC Update Unit - This unit examines the values of R and C in the message stored in the input buffer of the external bus interface unit. As mentioned above, the destinations for messages transmitted in the system can be predetermined. Thus, it is possible to use a simple routing scheme to get each message to the proper destination. The scheme I used involves the use of displacements in the horizontal direction (number of columns, C) and vertical direction (number of rows, R) between the location of the origin and the location of the destination. The receiver of a message first checks for nonzero values of R anc C, to determine whether the message should be forwarded to another node. If either is nonzero, the values are updated, and the message is forwarded. Since the values are examined when a message is received, the message will be sent one step when both values are zero. Thus, if a message is to be sent only to the nearest neighbor, both R and C are set to zero. If the message is to be sent farther, the displacements are set to nonzero values to indicate this, consistent with the fact that one step will be taken after both reach zero.

If either R or C is nonzero in the received message, the message must be sent to another node. The RC update unit updates the value of either R or C by the use of a small adder, to reflect the fact that the message has moved one step closer to its destination. It also computes the destination CP number, i.e., which CP is connected to the processing element which should receive the message. Both of these calculations are based on a predetermined routing scheme, so they can be performed quickly, without the need for a lot of computation or external information. If the message is to be sent on, the RC update unit signals this, and sends the message containing the updated R and C values, to the Internal Bus Interface.

Internal Bus Interface - When the RC update unit has determined that a message received by the external bus interface unit needs to be sent to another PE, this unit sends the message to another CP in the same processing element, which will then send it on to the neighboring PE. Communication between CPs in the same PE takes place over a shared bus, which I refer to as the internal bus. This bus has its own access controller.

To send a message, the CP asserts a request line, and waits for a "Granted" signal from the controller. Then it places the destination CP number (as received from the RC Update unit) and the message on the data lines, and asserts a signal to indicate that data is ready on the lines. All the CPs monitor the address lines to recognize when a message is being sent to them. When the address matches that of a CP, and the data ready signal is asserted, that CP must store the message in a buffer; it waits until this buffer is empty, then stores the message there and asserts an "Ack" signal, telling the sender that the data was received. At this point, the sender releases the bus.

Once the internal bus interface has received a message from another CP, that message must be sent to another PE by the external bus interface unit. That unit monitors the status of the buffer used by the internal bus interface, and forwards the message after it is received.

AU Interface - This unit receives data consisting of an x value (x_j) and a matrix entry ($a_{i,j}$) from the Associative Memory. It sends this value to the Arithmetic Unit. All the CPs share a single data path to the AU, and the AU uses this same path to send results to the CPs. In order to send data over this bus, the CP must request use of the bus. The AU monitors these requests, and signals one CP that it is granted access to the bus when it has a place to store the data values. Then the AU interface places the message on the data lines and asserts a signal indicating that data originating from a selected CP is present on the bus. It then removes the data immediately, on the assumption that the AU was ready to receive it when it issued the bus grant signal.

The same bus is used by the AU when it has a result to send to the CPs. In this case, the AU simply places the answer on the data lines, and asserts a signal indicating that data from the AU is on the the bus. All CPs must read in the data whenever this signal is asserted. It is not possible for the AU and the CPs to need to send data over this bus at the same time. To see this, consider the data dependencies of SOR iteration - the AU will not finish an iteration until it receives data from all the connected nodes, so it will not try to send out an answer while it is receiving data. However, once the AU receives the last data item needed to complete its iteration, the other PEs which could send more data will be waiting for the result of the calculation at this node in order to complete their own computations, so they will not have data to send to this PE. Thus, there can never be a conflict between a CP and the AU for access to this bus, nor can the buffer which the AU interface uses for sending and receiving messages be full when the AU sends a message to the CP. This simplifies the design of the interface.

Message Generator - When the AU interface receives a result from the AU, it signals this fact to the message generator. This unit generates new messages to send out to the other processing elements which need that result for their computations. It stores a list of the displacements to these nodes (i.e., the R and C values for the messages). It automatically generates a sequence of messages, one for each RC pair, and appends the node number (i) and R and C to the x value received from the AU interface. It sends these messages to the external bus interface to send on to the other nodes.

All of these units are capable of performing their functions without external assistance, so they can all work in parallel, as conditions require. Each is driven by its own controller, so that none has to wait for service from an external processor. Thus, these unit achieve a very high degree of parallelism in their operation. They only wait when forced to by external events, such as when a bus they need access to is in use, or a receiver for their data is not ready to receive. They communicate their status sufficiently to prevent any data from being lost, without any extraneous communication which would cause unnecessary delay.

Data Communication

A critical portion of this design is the processing of messages which must be routed to their destinations through several intermediate nodes. This is necessitated by the limited interconnections between PEs (i.e., direct connections to only the four nearest neighbors). Such a limited interconnection makes it possible for the sophisticated processing associated with each CP, as described above, to be incorporated on a single chip, and keeps the number of I/O pins required for each chip to a practical level. However, if this scheme takes too long to route messages to their destinations, it will limit the performance of the entire system.

Perhaps the best way to examine this processing is to trace the progress of a message as it is routed through the network. A few assumptions are required for this. I will assume that the external data bus linking CPs has a number of data lines equal to one third the number of bits in a message. I will trace the progress of a message which needs to be sent to a PE one unit to the left and one unit down. I will also assume that the intermediate processors which need to receive and forward the data are ready to do so, i.e., that they are not busy with similar processing on other messages (results of detailed simulations presented in Chapter 4 discuss the validity of this assumption). Rather than make any assumptions about the clock speed possible with the chip, I will discuss the operation in terms of the number of clock cycles required. For convenience, I will refer to the PE which originates the message as PE A, the PE which receives this message and forwards it on as PE B, and the destination of the message as PE C. I will describe the CPs as CP Up, CP Down, CP Left and CP Right, to indicate the direction to the neighboring PE with which the CPs communicate.

In order to send the message left one column and down one row, it must be sent out from CP Left of PE A. This would move the message the required number of columns, so the value of C should be set to zero to prevent sending the message additional columns to the left. The rows and columns of processors are considered to be numbered from the upper left corner, so that going down or right are positive displacements. Thus, to go down one row, the value of R in the original message would be set to 1. Then the progress of the message with each clock cycle would be approximately as follows (this process is illustrated in Fig. 7, with the numbers next to the arrows indicating the clock cycle at which the message arrives at that point):

0 - CP Left asserts request for external data bus.

1 - CP Left receives grant from CP Right of PE B.

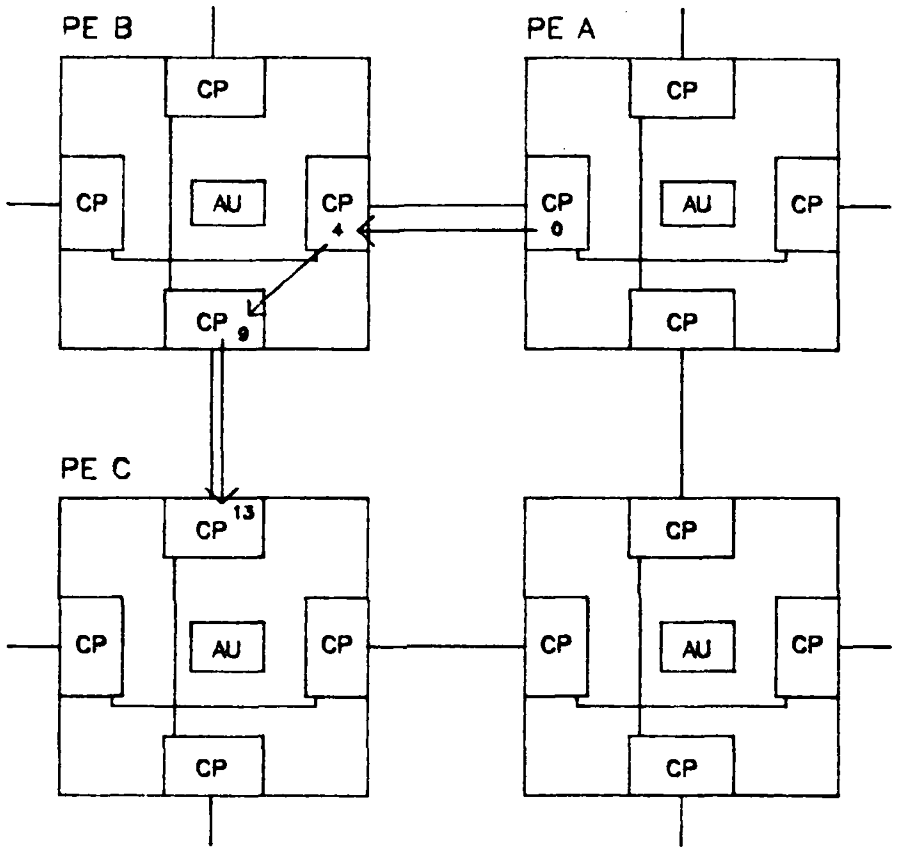


Figure 7 - Message Flow

2 - CP Left places first portion of message on data lines and signals data ready.

3, 4 - CP Left sends second and third portions of message to PE B. On receipt of third portion, external bus interface at receiving CP signals RC update and associative memory units that it has now data.

6 - RC update unit of CP Right, PE B, signals internal bus interface that message is to be sent to CP Down to for transmission to PE C. Since R was greater than 0, 1 is subtracted from R, and revised values of R and C (0 and 0) are sent to the internal bus interface unit along with the original values of $a_{i,j}$ and x_j.

7 - Internal bus interface requests access to bus.

8 - Internal bus granted. Interface unit places data on data lines and destination CP number on address lines and asserts data ready signal.

9 - Receiving CP (CP Down, node B) latches data into its buffer and asserts Ack signal, telling sender it has received the data. External bus interface detects that the receive buffer is now full.

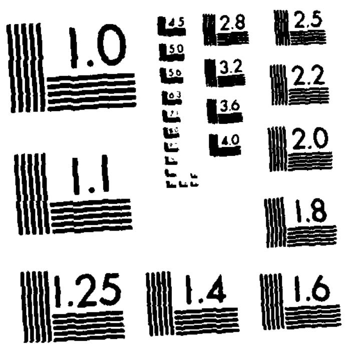
10 - External bus interface (of PE B, CP Down) moves data to its output buffer, and checks status of buffer full signal on receiver (CP Down is a bus master).

11 - 13 - External bus interface sends data to CP Up for PE C. It processes data in the same way as if PE B had been the originator of the message. Since R and C are both 0 in the message received, the message will not be forwarded.

From this discussion, it can be seen that under the assumptions described, it takes about 10 extra clock cycles to route data intended for a particular processor through an intermediate processor. Conflicts for the use of internal or external busses or more restricted external bus data widths would increase this delay, but the extent of this increase would depend on how often conflicts occurred, how long they lasted, and so on. Thus, accurate analysis of the effects of this delay on processing requires detailed simulation, in order to account for all the situations which can affect this performance. However, it is possible to consider some of the basic effects of this delay without knowing its exact magnitude.

Clearly, the impact of this delay will increase as the distance between PEs which must communicate grows. However, a direct addition of this number of clock cycles to the time required to complete an iteration does not accurately show the effect of this delay. There are several factors to consider:

AD-A171 256 A PARALLEL PROCESSOR FOR THE SOLUTION OF LARGE SPARSE 2/4
SYMMETRIC LINEAR SYSTEMS(U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH D A ARPIN11986
UNCLASSIFIED AFIT/CI/NR-86-124D F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

- All the data communication requirements are predetermined. Thus, this delay is added only once to the communication between two PEs; if one had to request the data, and then the other had to send it, the delay would be added in twice.

- For the typical regular finite element grid, as shown in Fig. 1, the data from all four nearest neighbors will be needed at each PE (except those on the edges of the grid). Once any data item arrives at a PE, it can begin computing. As long as the next item arrives before the first computation ends, the next multiplication will start as soon as possible. For example, if the multiplication requires 20 clock cycles, then once a data item begins multiplication, the computation for the next one can not begin until 20 clock cycles later, regardless of whether it arrives immediately or 15 clock cycles after the first multiplication starts. If a PE receives four data items at about the same time, and a fifth item somewhat later, the fifth item would have to be delayed by 80 clock cycles in order for this delay to have any effect on the computation time. This is equivalent to routing the data through about seven intermediate nodes.

- The extra message traffic caused by the need to route data over longer distances increases the probability that the processors and busses needed to route the data will be busy, not only for the long distance communication, but also for the communication needed to support the processing at the intermediate nodes. This is in the realm of second order effects, and very difficult to quantify analytically. This effect is best evaluated through the use of simulation.

- The partitioning of the PEs into autonomous Arithmetic and Communications units makes it possible for the AU to continue its computation while the CU handles the data to be routed to other nodes. Thus, the use of this type of communication does not detract from the computations at intermediate nodes (aside from the possibility of delaying the data they need slightly, as described in the previous paragraph). This parallelism between communication and computation is crucial to the effectiveness of this communication scheme. If the processor had to stop performing computations in order to route data to other nodes, this routing would have a major impact on the performance of the network. With both functions operating in parallel, the impact of communication delays on the computation can be minimized.

Based on these factors, this architecture appears capable of implementing SOR iteration efficiently for the type of problem I am interested in. As described above, the various functional units in each processing element function independently, affecting each other only when data dependencies or conflicts in access to other units arise. These interactions tend to be complicated, and problem dependent. Thus, to determine their effects, it is necessary to model the operation of the system in a way which takes them into account accurately. This is the subject of Chapter 4 of this dissertation. Since the evaluation of the performance of this architecture depends on simulation, I will defer the discussion of the capabilities and limitations of this architecture, and a comparison with other architectures, until after the presentation of simulation results.

Detailed Design

The remaining portions of this chapter cover the design of the various functional units of the Communication Processor in more detail. In this section, and in the section of Chapter 5 which describes the prototype, I try to maintain consistency in the names of the various registers and control signals used in the text, figures, and program listings. Technical limitations (e.g., the limited number of characters available for signal names in the logic analyzer, and the incompatibility of certain characters in different software packages used) make it impossible to maintain complete consistency.

Figure 8 shows the primary registers used within a CP to hold various data items which must be passed through the system. INBUF is the register used by the external bus interface unit to store messages received from the neighboring node. This data is used by several other units, and is retained in this buffer until they have completed their processing. AUBUF is used by the AU interface to store data which needs to be sent to the AU, and results received from the AU. RTBUF is used by the internal bus interface unit to store messages which are to be sent to another CP in this node for forwarding to a neighboring node. NMBUF contains new messages, generated by this CP as a result of receiving the next iterated value of x_i from the Arithmetic Unit. IBBUF is used by the internal bus interface to store data received from another CP in this node, which must be sent to a neighbor. OUTBUF is used by the external bus interface unit to store messages to be sent out over the external bus.

Most of the buffers contain the messages which are used to communicate data between nodes. Each message consists of values of a node variable (x), an index identifying which node the variable represents (i if the variable represents

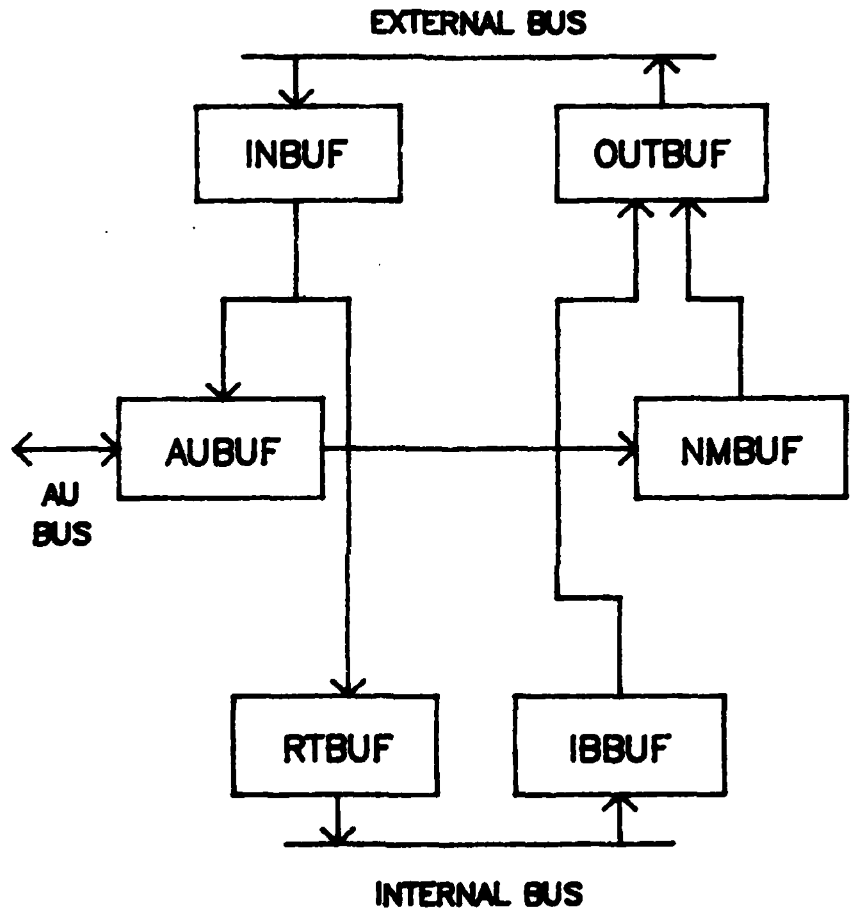


Figure 8 - CP Registers

the node in which the CP is located, or j otherwise), and the numbers of rows (R) and columns (C) of displacement in the grid of processors to the final destination of the message. While the architectural design of the system is not tied to the exact sizes of these data items, I have used the following estimated lengths for these variables: x - 32 bits, j - 16 bits (this number allows 2^{16} or 65,536 total processing elements in the system), and R and C - 6 bits each, for a total of 60 bits per message. The exception to this format is the AUBUF register. This register must hold the values of the variable x, and the associated matrix entry, which will be multiplied together in the AU. I have assumed that each of these items is 32 bits long, for a total of 64 bits in this register.

External Bus Interface

The detailed design of the external bus interface is shown in Fig. 9. The primary function of this unit is to communicate with the associated CP of a neighboring node. In order to do this, it must monitor the status of the neighbor, and also of many of the functional units within this CP.

I refer to the data lines between two CPs as the "external bus". In operation, either of the two CPs may

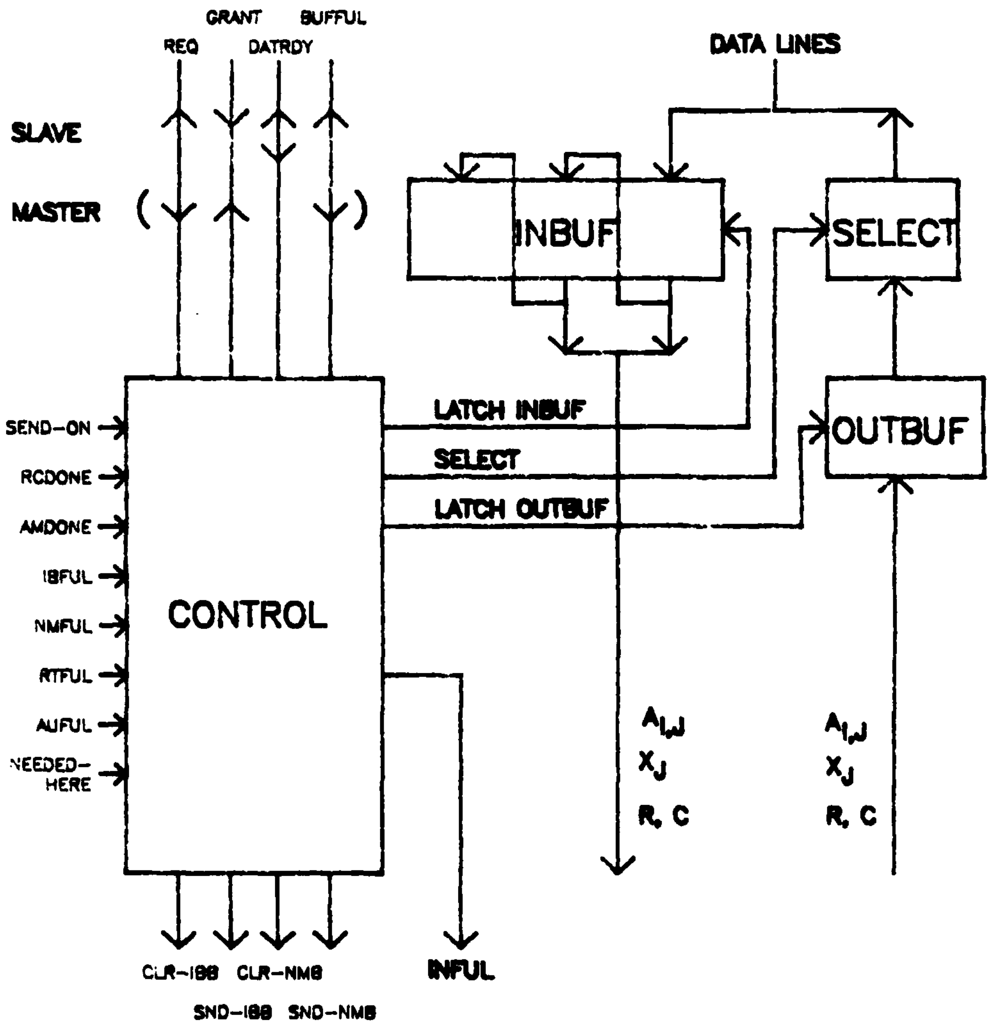


Figure 9 - External Bus Interface

need to send data over this bus at any time. Therefore, some method of controlling access to this bus is required. I provided for this control by designating some CPs as bus "masters", and the others as "slaves". This allows the CPs to coordinate the use of the bus and control the communication with four control lines, designated REQ, GRANT, DATRDY, and BUFFUL. Note the extra set of arrow heads shown in parentheses in Fig. 9. The top set of arrow heads illustrates the direction of control flow for an external bus slave CP, while those in parentheses indicate the direction for a master. Use of these signals is described below. The CPs designated for communication with the neighbors in the Down and Right directions are masters, while those for Up and Left are slaves.

When the master has data to communicate to the slave, it must first check the status of the BUFFUL (input buffer full) signal. The slave asserts this signal when its input buffer is full with data which it must retain, so the CP is not ready to receive data. The master must wait until this signal is clear before sending a message. Once the BUFFUL signal is clear, the master begins sending the message. It selects the first portion of the message to be sent and places those bits on the bidirectional data lines. When these lines are stable, the master asserts the DATRDY signal, telling the slave to read in the portion of a message on the lines. I have assumed that the delay to stabilize the data lines is less than half of a clock cycle, so the data can be placed on the lines and DATRDY can be asserted in one clock cycle. On succeeding clock cycles, the master changes its selection to place succeeding portions of the message on the data lines, again asserting data ready for each portion. It repeats this process until the entire message has been transmitted. The time required to complete this process depends on the number of portions the message must be divided into, which in turn depends on the number of bits in the message, and the number of data lines in the bus. I have shown connections on the INBUF (input buffer) which will route the portions of the message to the appropriate portion of the register as succeeding portions arrive. This structure could be implemented on a bit for bit basis, or contiguous segments of the message could be kept together, at the convenience of the chip designer, as long as the selector in the sending CP and the organization of the receiving INBUF matched.

When the slave needs to send data to the master, it asserts the REQ (request external bus) signal. When the master is ready to receive data, it asserts the GRANT signal, telling the slave to proceed with sending the message. From this point on, the process of transmitting data is the same as above. The slave may assert REQ at any time; the master will only assert GRANT when it is ready to receive data. If the master has data to send, it will send it, even if the slave has asserted REQ and the master is ready to receive it. Thus, this scheme prevents both CPs from trying to send data over the bus at the same time, and automatically resolves conflicts over access to the bus. It does these things very efficiently: assuming there is no reason why a CP may not send data, the master does not require any clock cycles to gain access to the bus, and the slave needs only one clock cycle after asserting REQ to receive GRANT and place data on the data lines.

Once the message has been received in INBUF, the controller for this unit sets a flag (INFUL) indicating to other units in this CP that a new message has arrived in this buffer. The other units which use this data report their status back to this controller, so it will know when all of them are finished with it. When the combination of RCDONE, SEND-ON, RTFUL, AMDONE, NEEDED-HERE, and AUFUL indicates that the data from INBUF either is not needed at a destination or has already reached there, the controller clears INFUL.

The other functions performed by this unit involve determining when other units have data which must be sent out by it, moving the data to OUTBUF, and informing the other unit that the data has been received. Thus, the controller monitors the NMFUL and IBFUL signals. When it has room in OUTBUF, it signals one of these units to send the data to OUTBUF. It latches this data in, then signals the source to clear the buffer which the data came from.

The controller thus responds to whatever data arrives. This allows a high degree of parallelism in operations. For example, if an item arrives in the internal bus buffer during reception of a message over the external bus, the controller can move this message to OUTBUF, without having to suspend the transmission or wait until it is complete. From the number of status signals received by this unit, and the number of control signals it sends to other units, it is clear that the functional units are interdependent. However, the dependency is entirely in terms of data - the operation of this unit does not depend on service from another unit to complete its tasks. This approach is used in all the functional units of my architecture, in order to allow maximum parallelism.

RC Update Unit

Figure 10 illustrates the configuration of the RC update unit. This unit examines the values of R and C in the message contained in INBUF to determine whether the message needs to be sent to other nodes in the network. If either R or C is nonzero, the message must be forwarded. This can be determined very quickly, using combinational logic, as shown in the figure. The processing beyond this

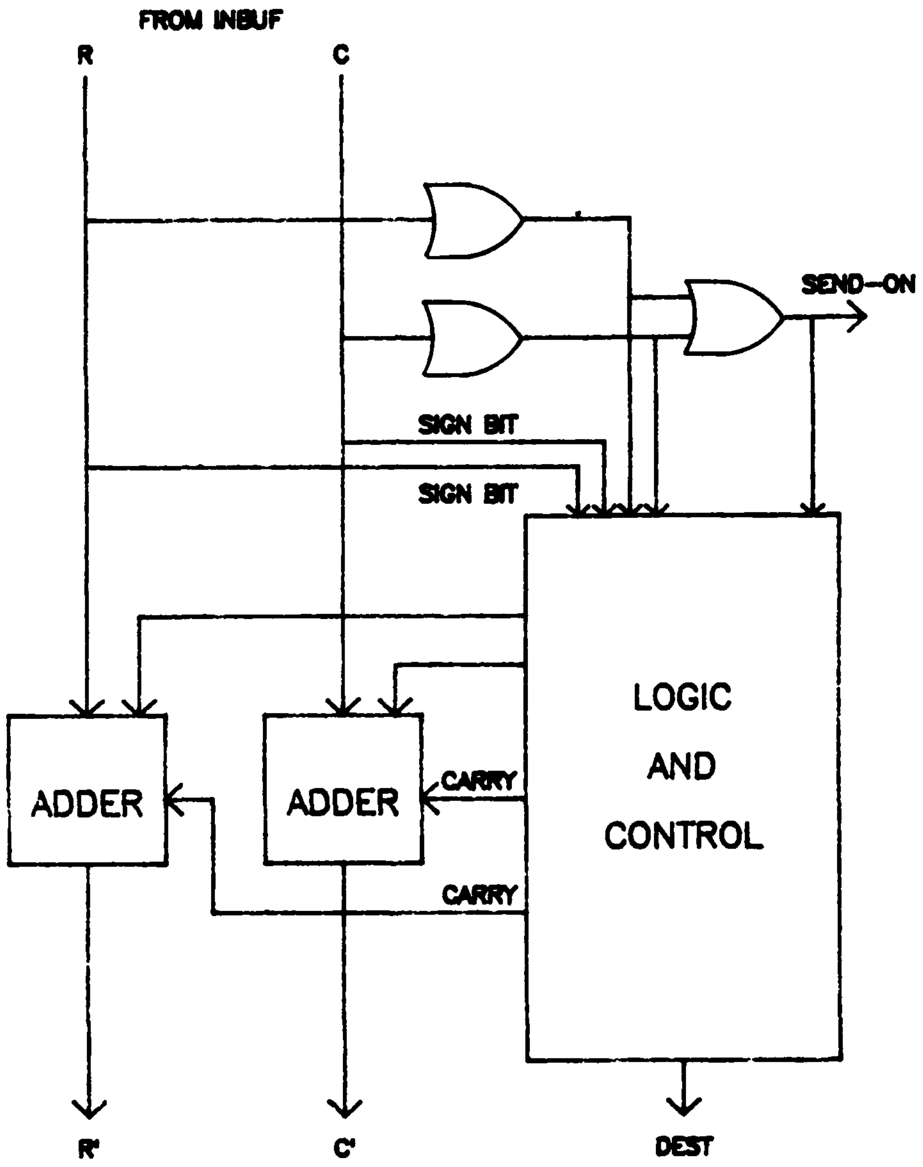


Figure 10 - RC Update

point is more complicated, and depends on the values of R and C and which CP this is. If either R or C is nonzero, two things need to happen. The value of either R or C needs to be updated, to reflect the fact that the message hs moved one row or one column closer to its destination. Then, the destination address for the internal bus (i.e., the destination CP number) must be determined. Both of these can involve choices which depend on which CP is doing the processing.

The basic routing scheme calls for a message to traverse rows and columns in a specified order. If the ultimate destination is directly to the left, right, up or down, then only R or C will be set, and messages will go through only pairs of Left and Right CPs or Up and Down CPs - no routing options are possible. To get to a destination which is in a diagonal direction, there are several paths of the same length. I decided that there was no particular advantage to any one choice, so the logic required to select the path could be simplified by always using the same path. Thus, the order to be followed in particular cases is as shown in Table 1.

Table 1 - Message Routing Order

Direction to Destination	Order
Up and Left	Left (-C), then Up (-R)
Down and Left	Left (-C), then Down (+R)
Up and Right	Up (-R), then Right (+C)
Down and Right	Right (+C), then Down (+R)

This routing method requires the CPs to update R and C and transfer messages over the internal bus as shown in the Table 2. In this table, the first two columns indicate the signs of the received values of R and C. The next four columns indicate how R and C are updated (e.g., +R indicates that the value of R is incremented, or -C that C is decremented), and to which CP the message is sent over the internal bus. The table does not show the case of R and C both zero, since the message is not forwarded in this case.

Table 2 - RC Update Processing

R	C	Up	Down	Left	Right
0	+		-C Right	-C Right	
0	-				+C Left
+	+			-C Right	
+	-				+C Left
+	0	-R Down		-R Down	-R Down
-	+		+R Up		
-	-				+C Left
-	0		+R Up		+R Up

The blank spaces in Table 2 indicate combinations of R and C which will not occur in this routing scheme. A small amount of combinational logic unique to each CP is needed, to determine which variable to update, whether it should be incremented or decremented, and which CP the result should be sent to. The updating of R and C is accomplished by the use of a small adder for each. Construction of two six bit adders should require only a modest amount of logic. Using this approach, a value can be updated simply by adding either 1 or -1 (in two's complement form) to one of them. I assumed that R and C would each be 6 bits long. This allows each message to be routed as many as 32 rows and 32 columns in either direction in the array of processors.

The inability of a PE to send messages to other PEs more than 32 rows or columns away limits the system to the solution of problems which can be mapped onto the grid of processors in such a way that no interprocessor message need be routed farther than this distance. The system is designed to exploit the locality of data inherent in the types of problems described in the previous chapters. Communications over much longer data paths are likely to cause excessive delays which seriously hinder the performance of the system. This is an example of the effect of designing a system to work well for a particular type of problem - the system will not work efficiently for other types of problems. Since the system is not designed to work for problems which require long communication paths, the inability to route data over such long paths is not a significant limitation in the design.

I have assumed that one clock cycle is sufficient for the combinational logic to determine the signs of R and C, and apply the appropriate values to the inputs of the adders. In one more clock cycle, the adders can produce the revised versions of R and C, denoted R' and C'. Thus, after two clock cycles, the controller sets the signal RCDONE, indicating that the values of R', C', DEST, and SEND-ON are valid. This total of two clock cycles matches the speed of the associative memory exactly, which is convenient, since both of them operate at the same time. Thus, neither the determination of whether the data in the message is needed at this node, nor the determination of what to do with the message, will produce a bottleneck for the operation of the system.

Associative Memory

The basic design of the associative memory is shown in Fig. 11. The purpose of this unit is to determine whether the data item in INBUF is needed by the AU for this node in order to perform its computation. As described in Chapter 2, it is possible to know before iteration begins which node values are needed at any given node, from the structure of the matrix for the linear system. Thus, for each CP, a list of the node numbers for which both (a) the value is needed at this node, and (b) messages would arrive at this CP, can be generated. Both conditions are required since some values needed at this node will arrive via other CPs, so these node numbers need not be included in the associative

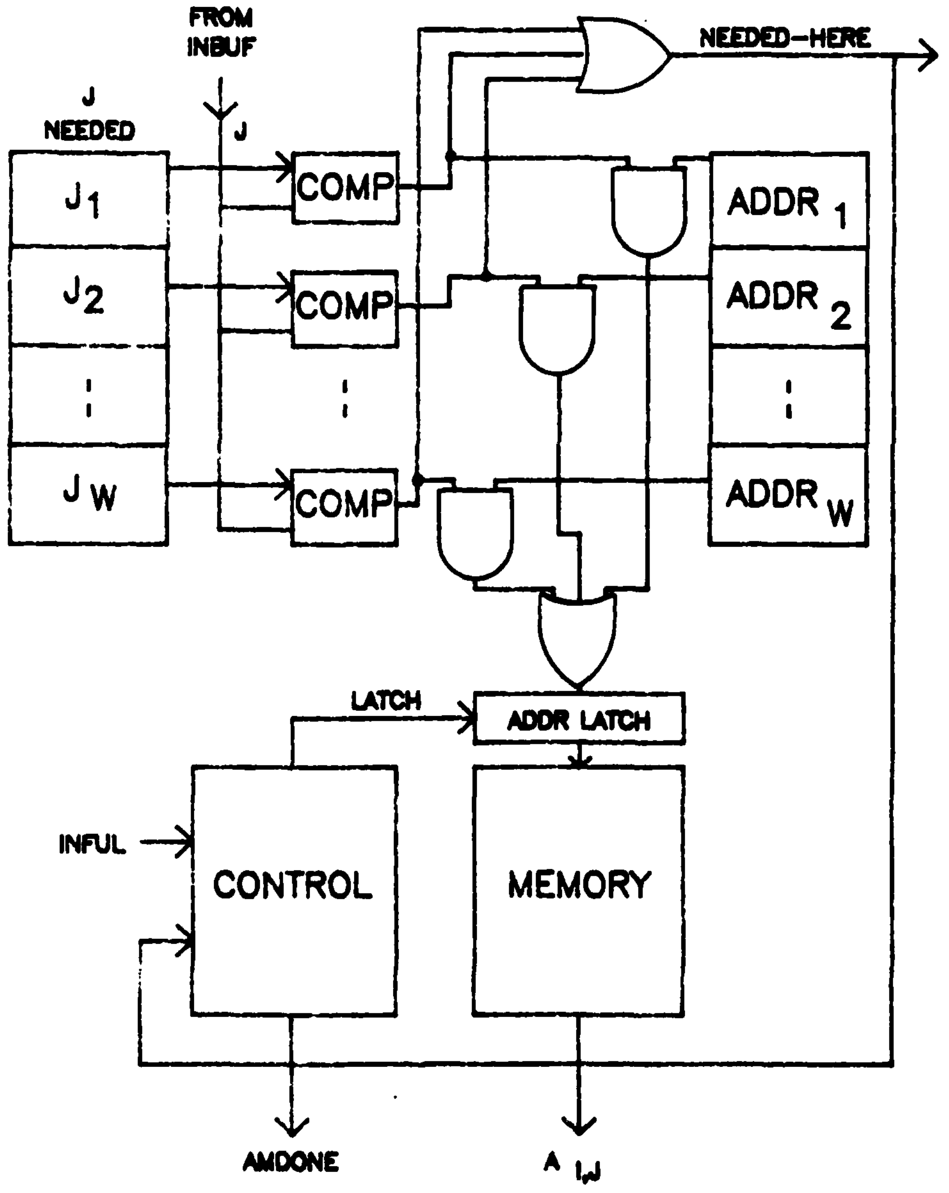


Figure 11 - Associative Memory

memory for this CP. This list is stored in the associative memory unit of the CP.

When a message arrives, it carries an index value with it. As shown in Fig. 11, this index value is simultaneously compared with all the stored values. If any of these comparisons indicates a match, the NEEDED-HERE signal is asserted, and an address for a small conventional memory appears at the input to the address latch for that memory. If the NEEDED-HERE signal and the INFUL signal are both set, the controller will cause this address to be latched into the address latch, within one clock cycle of the system. This address will cause the data value from that location in memory, representing the appropriate matrix entry, to appear at the output. This value will appear at the output of the memory within one additional clock cycle, after the address is latched. The controller will signal AMDONE at this time. Thus, within two clock cycles, this unit determines whether the data in the incoming message is needed by the AU at this node.

One more mechanism is required in this unit to handle messages properly, due to the routing of messages to distant nodes. In this routing scheme, the same data item may be passed to a node several times. For example, a message may need to be sent to a CP two columns to the left, and also to one one column left and one row up. From Table 1, it can be seen that both messages will pass the data item through the processor one column to the left. The associative memory must keep track of the messages received so that it does not forward the same data item to the AU twice for the same iteration (the AU processes whatever data it receives, so it would not detect this error). Thus, the associative memory must keep some record of the items sent to the AU, and not send items which it has already sent. This record must be reset at the end of each iteration (i.e., when the AU sends the result to the CPs).

The amount of hardware required to build such a memory grows rapidly with the size of the memory. The basic comparator, shown as a small box in the figure, requires B two input exclusive OR gates, and one B input OR gate, for comparing two values B bits long. For stored address values K bits long, K two input AND gates are required for each entry in the list of address values. Finally, for W words of storage, K OR gates of W inputs each are required to combine the address terms. Thus, it is important to keep K, B and W as small as possible.

The use of associative memory is practical for this application, since the matrix is sparse, and only a portion of the values to be sent to any node will arrive at a particular CP. Thus, a memory of approximately W = 8 words should be adequate for this purpose. This allows the stored address to be limited to K = 3 bits, since this is enough to address 8 words. B is set at 16 bits by the length of the node number, j. With each CP capable of receiving data from 8 other processors, each node can be connected to up to 32 other nodes (8 per CP times 4 CPs). Even complex models like that shown in Fig. 2b require each processing element to communicate with only eight neighbors. Thus, this number of entries in the associative memory should be adequate for the problems for which this system is intended. The construction of an 8 word associative memory should not present significant problems.

The two stage design of this memory is intended to reduce the total amount of logic required. If the matrix entry ($a_{i,j}$) corresponding to each word in the memory were stored where the address values are shown, 32 two input AND gates would be required for each word, instead of 3 gates as shown. Also, the OR gate shown combining the outputs of the AND gates would consist of 32 8 bit OR gates, rather than 3 such gates. Thus, storage of a three bit address saves a significant amount of logic. It should be possible to design the conventional memory with much less logic than this, since only three address lines need to be decoded, and 8 32 bit values need be stored. Direct implementation of an associative memory would save one clock cycle, since no intermediate latching of the address would be required. However, this function takes place at the same time as the operation of the RC update unit, which requires two clock cycles to function, as described above. Thus, no overall time savings would be realized by this increased complexity.

Note that this unit does not latch the data from the INBUF, but it does use the value of the variable subscript, j. Thus, INBUF can not be released if the NEEDED-HERE signal is asserted until the variable value (x_j) from INBUF has been moved to AUBUF.

AU Interface

This unit provides the communication between the CP and the Arithmetic Unit (AU). When the associative memory determines that a data value is needed by the AU, it places the variable and the matrix entry at the inputs to the AUBUF register, and signals that the data is needed by the NEEDED-HERE flag, as shown in Fig. 12. When this occurs, this unit latches the data into the AUBUF register, and signals AUFUL (telling the external bus interface unit that the data to be sent on has been latched, and need not be held in INBUF).

In order to send the data to the AU, the CP must gain access to the AU bus. First, it asserts REQAU (request AU bus). When the AU is ready to receive data from this CP, it signals AUGRANT to this particular CP. Then the CP places the data on the data lines, and asserts the CP-DATRDY (CP data ready) signal once the data lines are stable. It then immediately drops the data from the lines and the CP-DATRDY signal, without waiting for confirmation from the AU that it

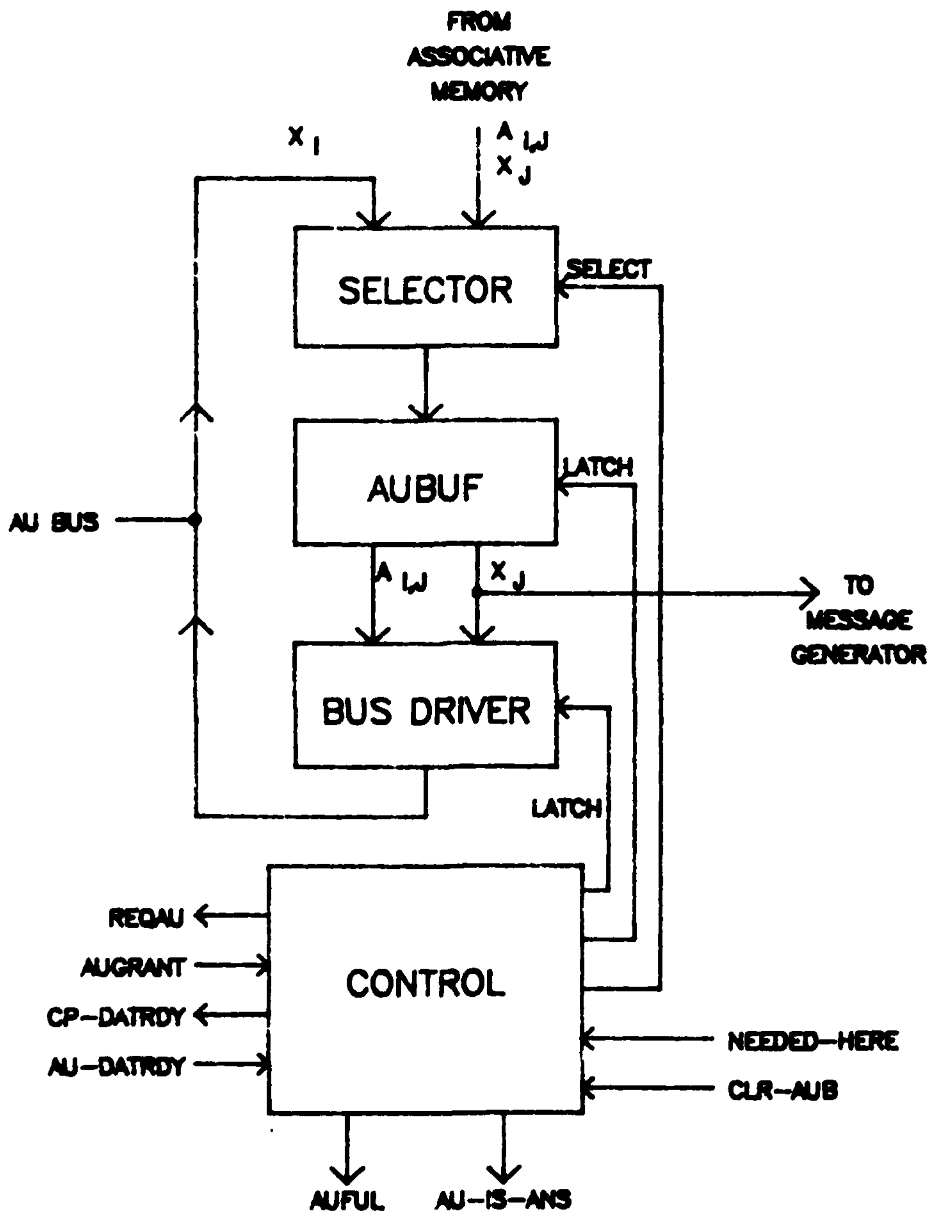


Figure 12 - AU Interface

received the data. This unit assumes that the AU received the data, since it indicated it was ready when it raised the AUGRANT signal. At this point, this unit clears the AUFUL signal.

When the AU completes the computations for one iteration, it must send this data to other nodes in the network. It does this by sending the data to all four CPs in the node simultaneously. To do this, it places the data on the data lines, and asserts the AU-DATRDY signal. All the CPs must store whatever data is on the AU bus when this signal is asserted. Since the AU is in control of access to the bus, there is no danger that it will try to drive the bus at the same time as a CP. Once the CP receives the data from the AU, it sets two flags: AUFUL and AU-IS-ANS. The second of these indicates that the data in the AUBUF register is the result of the computation, and must be sent to other nodes. The message generator unit uses this signal to begin its operation.

This explanation leaves open the concern of what will happen if the AUBUF register is full when the AU sends the result out. The design of the interface does not deal with this possibility, because it can not occur in the solution of a symmetric linear system. The AU sends out the answer only after it has completed its computations for this iteration, implying that it has received all the data from neighboring nodes required for this computation. Thus, the only data item which could be present in the AUBUF when the AU is sending out its answer would be an item which the AU required for its next iteration. However, a careful examination of equation 4 in Chapter 2 shows that the computations needed to provide the inputs for the next iteration at any node depend on the results of the previous iteration at that node. Therefore, once the AU receives the last data item needed to complete its computation, no new data items can arrive for that AU until the result from the AU has been sent to other nodes and they have completed their computations. Thus, it is not possible for the AUBUF to be full when the AU needs to send data there.

As with the external bus interface unit, this unit operates autonomously, and thus allows a high degree of parallelism in the operation of the CP.

Message Generator

Figure 13 shows the basic organization of the message generator. This unit monitors the status of the AU-IS-ANS signal. When this signal is asserted, this unit begins the process of sending out messages to the rest of the network to transmit the result obtained at this node. Each message contains the x value obtained by the AU, the node number of this node, which is stored in a small memory, and the routing information, R and C. The message generator for

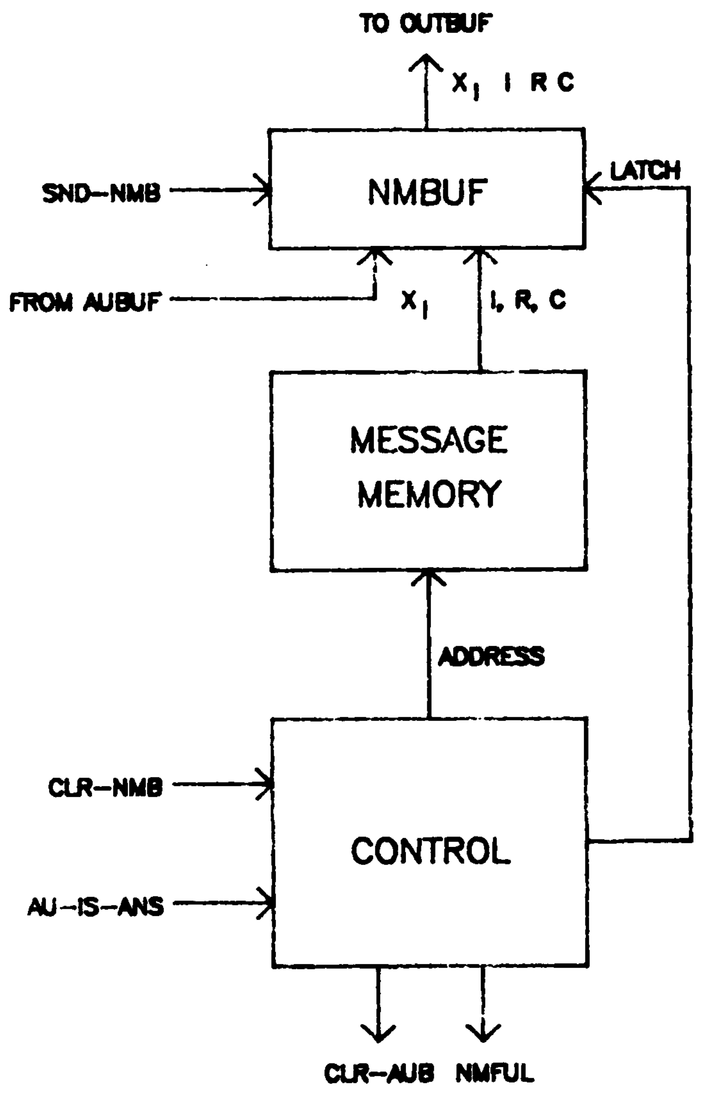


Figure 13 - Message Generator

each CP is programmed to send out only those messages which can be sent over the external bus connected to this CP (i.e., it does not generate messages which need to be sent over the internal bus).

Messages arriving at a CP can be used there, even if they are also sent on to other nodes. Thus, for the type of finite element grid shown in Fig. 2a, for example, the CPs in the processors assigned to the problem nodes in the center will need to send out only one message, even though some messages will carry data to two other PEs. Thus, it would be rare that a given CP would have to send out more than one message. This would only occur when the routing procedure described in the section on the RC update unit caused a message destined for a distant node not to pass through another node which also needed it. For example, if the data from node A is needed at node B, one unit down and one unit right, and also at node C, two units down and two units right, the message sent to node C would proceed two units to the right first, and then down to node C, bypassing node B. In this case, two separate messages would need to be sent. On the other hand, some CPs associated with processors on the edges of the grid need not send out any messages at all.

Thus, each message generator needs to send only a small number of messages. At the beginning of execution, the message memory shown in Fig. 13 would be loaded with i, and the RC pairs it needed to send. Then, when the controller detects AU-IS-ANS, it cycles through this memory, placing the stored values of i, R and C, and the value of x from the AUBUF, in NMBUF (the new message buffer). For each such message, it signals NMFUL, and waits for the external bus interface to remove the data and send it out. When the external bus interface unit has latched in the data, it signals CLR-NMB (clear new message buffer). Then the message generator repeats this cycle until it has sent all the required messages. At this point, it signals CLR-AUB to the AU interface unit, signifying that it has finished using the data in the AUBUF register, so the data can now be released.

Internal Bus Interface

Figure 14 illustrates the major components of the internal bus interface. Messages are sent between CPs over this bus whenever a message arriving at a CP has a nonzero value of R or C. This interface monitors the status of RCDONE and SEND-ON - when both of these are set, the data present at the inputs to the RTBUF (re-transmit buffer) must be sent to another node. The controller causes these values to be latched in to RTBUF, and also latches the value of DEST into a small buffer. Then it signals RTFUL, telling the external bus interface that data which needs to be sent

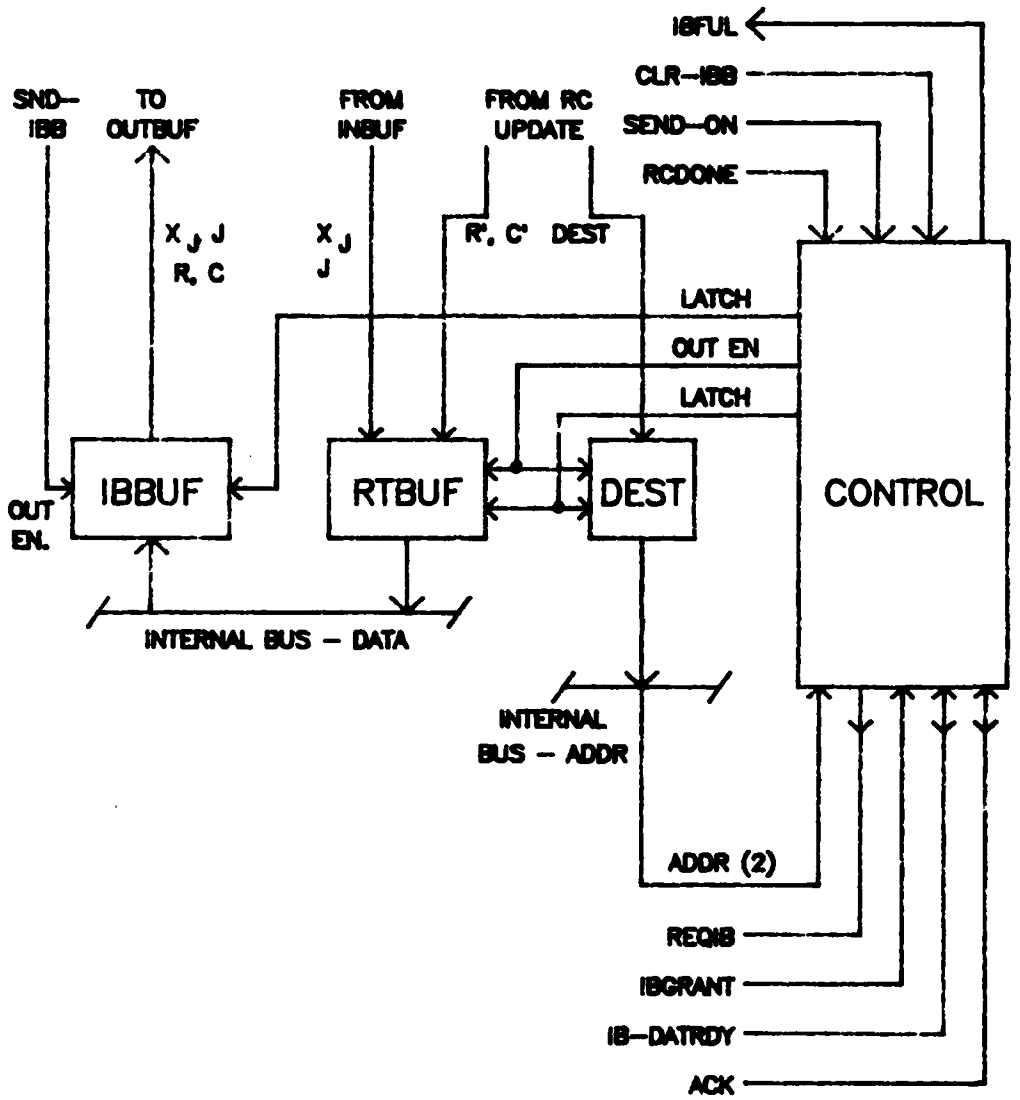


Figure 14 - Internal Bus Interface

on has been latched, and need not be held in INBUF any longer.

To send the data to another CP, the interface must first gain access to the bus. Unlike the external bus, the internal bus is shared by several users with equal priority. Thus, the bus requires a central controller, to handle the case in which several CPs want to use the bus at the same time.

The CP which desires to send data over the bus first asserts a REQIB (request use of internal bus) signal. When the bus is available, and the controller decides to allocate it to this CP (based on a round robin scheme), the controller signals IBGRANT to this CP. The CP then places the data in RTBUF on the data lines and the DEST on the bus address lines, and asserts IB-DATRDY (internal bus data ready). All CPs monitor the address lines. When one detects its address on these lines and the IB-DATRDY signal asserted, it knows that the data on the lines is for it. It must store this data in its IBBUF (internal bus buffer). But this register could be full of data which can not be overwritten yet. In this case, the receiving CP will simply wait until the external bus interface removes the data from IBBUF. At this point, the receiving internal bus interface will latch the data from the data lines into IBBUF. Then it will assert the ACK signal, acknowledging that it has received the data. The sender must hold the data, address, and IB-DATRDY signals until it receives this ACK signal. Then it drops these signals, and clears the RTBFUL flag, since the data in RTBUF is no longer needed.

Data must be held in the IBBUF register until the external bus interface has moved it to OUTBUF. It does this by asserting the SND-IBB signal, enabling the output of data contained in IBBUF. Once it has latched in this data, it signals CLR-IBB, telling the internal bus interface that the data is no longer needed. At that point, the internal bus interface can clear IBFUL, and is ready to receive more data. However, the external bus interface also receives data from the NMBUF (see message generator above), and gives priority to data from that source. When the AU finishes iteration, the message generator may send several messages in succession to the neighbor. During this process, data being routed between two other nodes via this node could still arrive via the internal bus interface. Such data items would have to wait in the IBBUF of the receiving CP until the external bus interface was ready to send them on. Thus, it is possible that a second message could arrive at the internal bus interface before the previous one had been removed. Hence, the ACK mechanism described above is necessary to avoid the loss of data sent over the internal bus.

Summary

This chapter has described the architecture which I have devised for the rapid parallel execution of SOR iterations. I believe that this architecture can function very efficiently for this application. The architecture is designed to exploit parallelism at many levels - with separate processors for each variable allowing parallel computations, separate arithmetic and communications units in each PE allowing communication and computation to take place in parallel, and separate activities happening in parallel within each communication processor, to make the overall operation as fast as possible.

With so many things happening at the same time, it is impossible to give exact estimates of the performance of this system just by simple analysis. There are many units interacting, with chances for conflicts between them. Thus, to determine the performance obtainable from this system, I used a detailed computer simulation. The next chapter describes this simulation, and the results I obtained from it.

CHAPTER FOUR - SIMULATION

One of the difficulties with any computer design is obtaining a meaningful measure of the performance of the system. Many factors can affect system performance, so even straightforward changes in a system do not always have the expected effects. For example, one might expect that doubling the clock speed of a microcomputer system would double the speed of processing. But if this change is implemented with no other adjustments to the system, the increased processor speed might cause the CPU to access the memory bus more frequently, thus reducing the amount of time available for peripheral devices to access the memory. For applications which required large amounts of such I/O, this conflict could limit the performance of the revised system to a fraction of the expected gain.

This problem is even more severe in parallel processors. Their performance is highly dependent on the nature of the problem and how effectively the program makes use of the processing resources available. A common approach to describing processing speed is to find the time required for one processor to compute an operation, invert this number to get a number of operations per second, and then multiply by the number of processors in the system. Thus, for example, the MPP processor can be said to execute over six billion additions (of 8 bit integers) per second [Potter, 1983]. However, the user of such a processor is interested in how fast his applications will run, not how many computations per second the machine can execute. Good performance for realsitic applications can only be obtained by careful design of the system to insure that there are no bottlenecks to limit the speed of the machine for the actual computations of interest.

If the processors in a parallel system run autonomously, it can be very difficult to estimate the performance of the total system analytically. It is not sufficient to simply find the amount of time required for each subtask and add these times together, since their operation may overlap, and the operations may be interdependent. Yet performance estimation is important for two reasons: to achieve a more efficient system design, and to determine the capabilities of the system once it is designed without physically implementing it. These requirements call for a tool which will allow the designer to model his system in enough detail so that he knows what is happening in the operation of the system, he can measure the effects of changes in the design, and he can estimate the performance achievable with the system for specific applications.

In order to meet these requirements, I constructed a detailed simulation model of my proposed architecture. I wrote the modeling program in the SIMULA language. This program modeled the functions of the various portions of my design, and their interactions, in detail. I used the model to investigate and compare various design alternatives, in order to make the design as efficient as possible. Then I used it to determine the performance available from the design for various types of problems.

This chapter gives a brief description of the SIMULA language, since the language has some unique features for my application. It presents a detailed description of the model I constructed, including its capabilities and limitations. Then it describes how I used the model to refine the design of the system and evaluate its performance, and gives the results of the performance analysis.

SIMULA

The SIMULA language was designed for the simulation of event driven systems [Birtwistle et al., 1973; Franta, 1977]. The language is based on ALGOL, with extensions to provide for the specific requirements of simulation. The basic concepts added to ALGOL to make SIMULA are classes, objects, queues, and hierarchies.

A "class" is somewhat analogous to a procedure in ALGOL. It may contain parameters, variables, procedures, and sequences of instructions to be executed; these are referred to as "attributes" of the class. The programmer may then create many instances of such a class, known as "objects". Each object might then differ in some attribute from other objects in the same class, such as the value of one parameter. The attributes of an object are available for use by other parts of the program, even after the execution of the object is finished.

A simple example of these concepts might be a class to simulate the operations performed by an automated teller machine used by a bank. Formal parameters for such a class might include the location of the machine and the networks it can communicate with (i.e., which cards it will accept). Variables could include such things as the amount of cash available for withdrawal, and the total amount of deposits for different institutions. Procedures would include the steps necessary to establish access rights after a card is inserted, conduct transactions, and process individual requests. All these attributes might be included under a class called "class ATM". In simulating a network of such machines, the programmer could then generate several instances of class ATM, to model the operation of specific machines in the network.

The concept of "hierarchy" allows classes to be created as subtypes of established classes. Class declarations can be concatenated together, resulting in a class with a combination of the attributes of the individual declarations. Thus, in the example above, the addition of a new type of machine to the network, with one extra function not available in class ATM, could be included in the model by adding a new class containing only the new procedure, and making the new class a sub-class of class ATM. The language has provisions for resolving conflicts which arise if different levels of classes use the same variable names, etc.

Included in the language are several general classes with attributes which are useful for different types of classes. Users can include these attributes in their own classes, in order to simplify such operations as list processing, event scheduling, and data input and output. These classes are known to the language, so users can declare their classes to be sub-classes of them without having to list the attributes of the system classes in their programs.

These system classes include extensive facilities for establishing, maintaining, and using queues. Any object can be placed into a queue, as long as it includes the necessary attributes, such as the pointers used in keeping track of the objects in the queue. Queues can be used in many ways, such as passing messages or sorting data records.

The system classes also provide for simulating the flow of time in the operation of the model. For example, the ATM might require ten seconds to print the receipt for a transaction. In simulating this, the object assigned to that machine can not simply advance the clock by ten seconds, since it has no way of knowing what other events should occur during that time. Thus, some means for simulating the passage of time is needed, which can model several things happening in parallel. A system class, called class "process", allows objects to suspend their own execution for a certain period of simulated time, and then be restarted automatically. When a process class object (or simply a process) needs to simulate the time required to complete an action, it places a notice in an "event queue", indicating when it should resume its processing, and then suspends its operation. A separate utility keeps these notices in order, and keeps track of the simulated time. When a process suspends its operation, this utility determines the time that the next event is scheduled to occur by looking at the first entry in the event queue, advances the simulated time to that time, and activates the appropriate process. If several processes are scheduled for the same time, they will be executed at that simulated time, but not necessarily in any particular order. Objects in the event queue can be rescheduled, cancelled, and so on as necessary during the simulation.

The SIMULA language is thus very well suited to simulating the operation of a collection of processors operating in parallel. It is capable of simulating each processor in any desired level of detail, and keeping accurate track of the interactions between processors.

Model

Based on the above considerations, I decided to model the operation of my designed architecture in SIMULA. The basic unit of time in this model is the clock cycle - all operations are modeled at the level of detail of what would happen for each individual clock cycle. This allows me to trace the operation of the processor in great detail, and get detailed performance measures, without having to make any assumptions about the clock speed which could be achieved (based on gate delays, bus loading, ripple carry propagation times, and so on).

Since the algorithm is iterative, I decided to make performance measurements in terms of the number of clock cycles required to complete one iteration. This was necessary, since the runs to complete the entire iterative process would have been prohibitively long. This also allows for simple extension of the results obtained to the case in which an arbitrarily specified number of iterations is required. Since the times are in terms of clock cycles, it is easy to convert these results to times for a given clock speed. All the factors which would cause the performance to fail to scale up with clock speed have been considered in the design of the units. Only the speed of the data interface to the host can limit this scaling - I will discuss this issue in Chapter 6.

The model uses separate processes (process classes) for each of the components of the communications processor, and for the AU. Queues are provided for passing messages between nodes modeled. Formal parameters associated with the processes indicate the position of the processor within the array, and other information such as whether the external bus interface is a master or a slave. Variables within the processes keep track of status, which variable the node is repsonsible for computing, the destinations for messages to be sent out, the indices of variables needed from other nodes, and so on. The procedures for each process handle all the processing which each functional unit would perform in the actual system, such as sending and receiving messages, determining the routing and disposition of messages received at a node, generating new messages to convey the results of computations to other nodes, and so on.

Some of the processes used do not model the associated operation in great detail. For example, the Arithmetic Unit model does not keep track of every operation in that unit. Instead, it uses certain assumptions about how many clock cycles different operations require (e.g., to store the result of an iteration and signal the CPs that the data is ready), without going into all the details of those operations. This does not cause any significant degradation in the accuracy of the simulation, since the functions which are modeled in this way do not interfere with each other in ways which would cause their time of execution to vary.

In other areas, the system is modeled in a great deal of detail, down to the level of what happens on each clock step - storing data in registers, asserting and dropping signals, waiting for acknowledgement, and so on. This approach is used for those parts of the system (i.e., the communication processors) which interact significantly.

Both types of models can use externally specified parameters for their operation. As mentioned above, the speed of the multiplier is not assumed in my design. Thus, there is a parameter in the simulation to specify how many clock cycles a multiplication takes. Once a multiplier starts an operation, it is marked "busy" until this time expires; while the multiplier is busy, another multiplication can not begin at that node. In another part of the model, a parameter is used to specify how many portions a message must be divided into for transmission over the external bus. However, unlike the multiplier busy case, each step of the transmission over the external bus is modeled separately. This usage of parameters makes the model simpler, and also more flexible. It is simpler, because it is not necessary to model in detail the parts of the design that have little or no impact on performance. It is more flexible, in that it is easy to submit several runs to determine the effect of changing parameters on the overall performance of the system.

In the construction of this model, a problem arose in using a serial processor to simulate the actions of a parallel array. In the actual system, many things happen at the same time, i.e., on the edges of the system clock. In SIMULA, all these things are modeled as happening at the same time by placing the notices in the event queue at the same time. However, as mentioned above, this does not insure any particular order of execution of the associated processes. But this can have an effect on the operation of the model if the output of one process provides input for another. For example, consider two edge triggered D flip flops (FFs) connected to a common clock, with the output of one connected to the input of the other. On a clock pulse, the second FF will respond to the old output of the first one. If a simulation of this simple system executed the process for the second FF first, the model would be accurate. But if the simulation executed the function of the first FF, then read that output as the input to the second, it would store the wrong value in the second one. In a highly parallel system, there are many such dependencies to consider at each clock cycle. SIMULA is not designed to take care of such situations automatically.

The solution to this problem consisted of breaking each clock cycle into two halves. On the first half cycle, all units read in any status information from other units which will affect their action. They store this information in "current status" variables, designated by names beginning with the letter "c". No unit may change any of its outputs during this first half cycle. Instead, each unit holds for one half clock cycle, until all others have read in the status at the beginning of the cycle. Then all the units are reactivated, complete their processing based on the status they read in previously, and change any of their outputs as necessary. No unit reads any status information on this second half cycle, so the order of execution of processes scheduled for the same time does not affect the results of the processing.

The model is designed to handle arrays of processors of any size, just as with the actual system. Just as in the actual design, the allocation of processors to nodes in the problem is done in software, so the nodes can be moved to different processors to determine the effects of this move on performance. The program allocates as many objects of each class as necessary to simulate all the hardware in the system (e.g., enough external bus interface units, associative memories, RC update units, and so on to construct four CPs for each processor). There are no inherent limits in the model for the size of array which can be simulated. However, there are practical limits to the model, caused by trying to simulate the operation of many parallel functional units on a single processor. As the size of the processing grid grows, and other parameters change to values which would slow down the operation of the actual system, the program gets bigger and/or slower to execute. This limits the operation of the model to problems of small size. For example, a problem using a grid of 3 rows and 5 columns and simulating roughly 900 clock cycles requires approximately 40 minutes of CPU time on a DEC System 20 computer. Thus, most testing had to be restricted to small models, with a small number of tests on larger models to determine how the results scale to larger problems.

The program is designed to be flexible in the amount of data it provides. In order to provide detailed outputs for debugging and design verification, it includes numerous output statements. However, these produced far more output data than is needed for most runs, which are primarily concerned with the overall performance level of the system. A print level parameter in the program makes it possible to select the amount of detail in the printed output to suit the purposes of the particular test run.

In operation, the model is controlled by the contents of four input files, denoted "nodeno.run", "dests.run", "needs.run", and "parms.run". The parms.run file contains a description of the numbers of rows and columns in the grid of processors, the parameters to be used as described in the previous paragraph, and controls for the execution such as the amount of printed output desired and the number of iterations to be completed. File nodeno.run provides the allocation of problem nodes to the grid of processors, by listing the variable numbers in row and column order. File dests.run provides the routing information for messages sent out from each CP in the system. For each CP, it specifies the number of messages to be sent, and the R and C values to be used with each. For convenience, these are input in the order of the rows and columns of the processor grid, rather than the order of the node numbers. This allows different node ordering schemes to be tried without having to change this file, as long as the pattern of communication is not changed. Finally, file needs.run specifies for each node of the problem the neighboring nodes from which values are needed. These items are stored in order of problem node number.

A listing of my simulation program, along with sample control files, is provided in Appendix C.

Problems Simulated

Most of my use of the model described above was in the area of refining the design. The program includes many status reporting commands, which allow tracing the operation of the system in detail. This allowed me to tailor the features of the system to the needs of the problem.

One example of this tailoring was in the external bus interface unit. I was concerned that slow functioning of this bus could cause data to back up, and thus interfere with the operation of other units (e.g., the internal bus interface would not be able to receive data, so the bus could get locked up, if the receive buffer stayed full too long). To prevent this, I originally included a first in, first out (FIFO) queue for messages to be sent out over the external bus interface. However, I had no way of knowing how large this queue should be, or even whether it was needed at all. By simulating the operation of the system, and monitoring the status of this queue, I was able to determine that the queue almost never contained more than one message, and even when a second message arrived before the first was removed, this situation lasted only a very short time. Thus, I determined that this FIFO was not required.

I also used the simulation to modify my original design for the internal bus, connecting the CPs of one processor. I originally designed separate busses between individual

CPs, operated in a master/slave relationship similar to that used for the external busses. This allowed some operations to take place in parallel, e.g., CP Up could send a message to CP Down, and CP Left could send a message to CP Right, both at the same time. On the other hand, it was not possible to provide direct links from each CP to every other one, so some messages had to be sent over two links (e.g., to send a message from Up to Right). I later considered replacing this system with a single, shared bus. This simplified the processing of the messages at each node, but it was not clear what the impact on performance would be, since this change would eliminate the parallel sending of messages, but also eliminate the need to send some messages over two separate links. Thus, I ran detailed simulations for both schemes and compared the results. These runs used the problem illustrated in Fig. 16a, described below. Table 3 summarizes the average number of clock cycles required for each iteration with the two different internal busses, assuming 20 clock cycles are required for multiplication, for several different assumed numbers of external bus cycles required to transmit all the bits of a message. This study led me to select the single internal bus, since the performances were virtually identical, while the single bus scheme was simpler to implement, and easier to expand to a three dimensional grid (see Chapter 6).

Table 3 - Performance vs. Internal Bus Configuration

External Bus Cycles	3 Independent Internal Busses	1 Shared Internal Bus
1	288	304
3	330	336
5	367	375

My other use of the simulation program was to measure the performance of the system once the design was stabilized. Even with a detailed simulation program, this area is problematic. The operation of the system depends on the details of the problem to be solved. In particular, each processor in my system must perform the multiplications and additions associatrd with the variables from nodes which are connected to that node in the original problem. Thus, if a node is connected to more neighboring nodes, the processor assigned to that node must perform more computations. Also, messages to neighbor nodes must be routed through intermediate nodes in the network. Thus, if nodes which must communicate values are farther apart, the messages will take longer to arrive, and the nodes which must carry these messages will be burdened by a higher traffic load than they otherwise would. In addition to these considerations, there are many options for the parameters associated with the performance of each processor, such as the speed of the multiplier. Finally, since the solution scheme used is iterative, the total solution time depends on the number of iterations required, which in turn depends on the problem being solved. Because the simulation program took so long to execute, it was not feasible to submit a large number of test runs, testing many combinations of parameters on many networks for many different problems, in order to get exhaustive data for analysis.

In order to gather reasonably reliable performance data in this situation, three basic types of problems were used. For the overall performance measures, the problem selected shouls give realistic measures of performance, rather than the most optimistic results. The best performance for this system is in solving problems which result from the finite difference method. In this method, each node is connected only to its four nearest neighbors, matching the interconnections between processors in the ParSOR architecture exactly. Thus, the communication between nodes is always done directly, with no routing of data through intermediate nodes. With only four nodes connected to each node, the number of multiplications required is minimized. These factors combine to allow such problems to run very fast on this system. Finite element method problems with triangular elements require more communication and computation, and thus run slower. As the number of nodes interconnected increases, the network slows down. Performance estimates should be based on problems of reasonable complexity, in which the nodes do not necessarily map perfectly into the processor grid, and in which some longer data paths are required.

For this purpose, a problem of 15 nodes was devised, with the nodes arranged in three rows of five nodes each. These nodes represent the corners of triangular elements, except two additional connections have been added between more distant nodes, as shown in Fig. 15. Thus, this model does not exactly match a real finite element model, but instead combines the factors of longer data paths and extra connections between nodes in a compact form. This model shows the effects of data dependencies between nodes 1 and 10 and between nodes 6 and 15 slowing down the processing, and thus shows slower execution than cases without these two extra connections. This model was used to compare the effects of various combinations of external bus bandwidth and multiplier speed on the overall processing speed of the system.

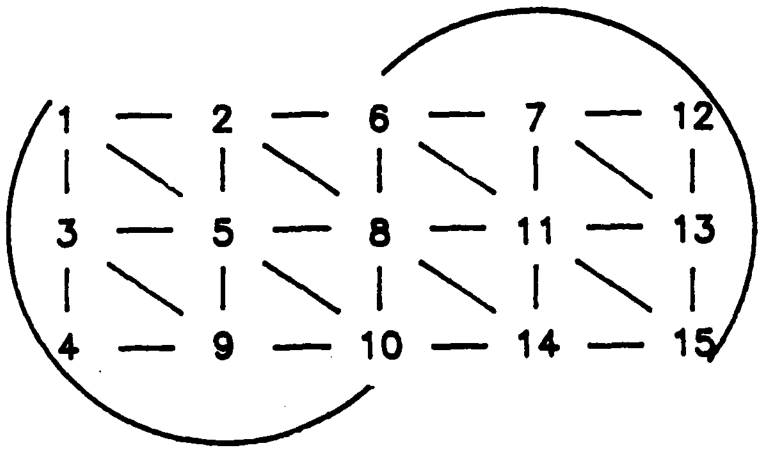


Figure 15 - 15 Node Test Problem

The second model used had 16 processors in a four by four grid. The elements are triangular, so the connections are like Fig. 15, but without the two extra connections shown there. This model was used to gain performance measurements for this more optimistic case, and also to compare the effects of the diagonal node ordering used for many early simulation runs with the more efficient "node coloring" ordering. Figure 16 illustrates the two models used for this comparison. Figure 16a shows the model using diagonal ordering, while Fig. 16b illustrates a three color node numbering scheme for this problem (this scheme is described in Chapter 2).

The third model used involved 36 nodes, in a six by six

$$\begin{tikzcd}
1 \arrow[r, dash] \arrow[d, dash] \arrow[dr, dash] & 2 \arrow[r, dash] \arrow[d, dash] \arrow[dr, dash] & 6 \arrow[r, dash] \arrow[d, dash] \arrow[dr, dash] & 7 \arrow[d, dash] \\
3 \arrow[r, dash] \arrow[d, dash] \arrow[dr, dash] & 5 \arrow[r, dash] \arrow[d, dash] \arrow[dr, dash] & 8 \arrow[r, dash] \arrow[d, dash] \arrow[dr, dash] & 13 \arrow[d, dash] \\
4 \arrow[r, dash] \arrow[d, dash] \arrow[dr, dash] & 9 \arrow[r, dash] \arrow[d, dash] \arrow[dr, dash] & 12 \arrow[r, dash] \arrow[d, dash] \arrow[dr, dash] & 14 \arrow[d, dash] \\
10 \arrow[r, dash] & 11 \arrow[r, dash] & 15 \arrow[r, dash] & 16
\end{tikzcd}$$

A

$$\begin{tikzcd}
1 \arrow[r, dash] \arrow[d, dash] \arrow[dr, dash] & 7 \arrow[r, dash] \arrow[d, dash] \arrow[dr, dash] & 12 \arrow[r, dash] \arrow[d, dash] \arrow[dr, dash] & 2 \arrow[d, dash] \\
8 \arrow[r, dash] \arrow[d, dash] \arrow[dr, dash] & 13 \arrow[r, dash] \arrow[d, dash] \arrow[dr, dash] & 3 \arrow[r, dash] \arrow[d, dash] \arrow[dr, dash] & 9 \arrow[d, dash] \\
14 \arrow[r, dash] \arrow[d, dash] \arrow[dr, dash] & 4 \arrow[r, dash] \arrow[d, dash] \arrow[dr, dash] & 10 \arrow[r, dash] \arrow[d, dash] \arrow[dr, dash] & 15 \arrow[d, dash] \\
5 \arrow[r, dash] & 11 \arrow[r, dash] & 16 \arrow[r, dash] & 6
\end{tikzcd}$$

B

Figure 16 - 16 Node Test Problems

arrangement. This model was used only to determine whether scaling up the size of the grid, without changing the interconnection patterns, would slow down the processing. Thus, the node numbering and interconnections are similar to those in Fig. 16a.

For many of the test runs, the first iteration does not provide an accurate picture of the time to complete an average iteration. This is because each processor uses assumed values for higher numbered nodes to get the iteration started, and also because the iteration must propagate through the entire mesh of processors in some cases. Therefore, my test runs included three iterations; for the overall performance measure, I discarded the time for the first iteration, and averaged the times for the second and third (they typically differed by only a few clock cycles due to transients in the transition from the first iteration which used some assumed values, and the later iterations which require the communication of all values).

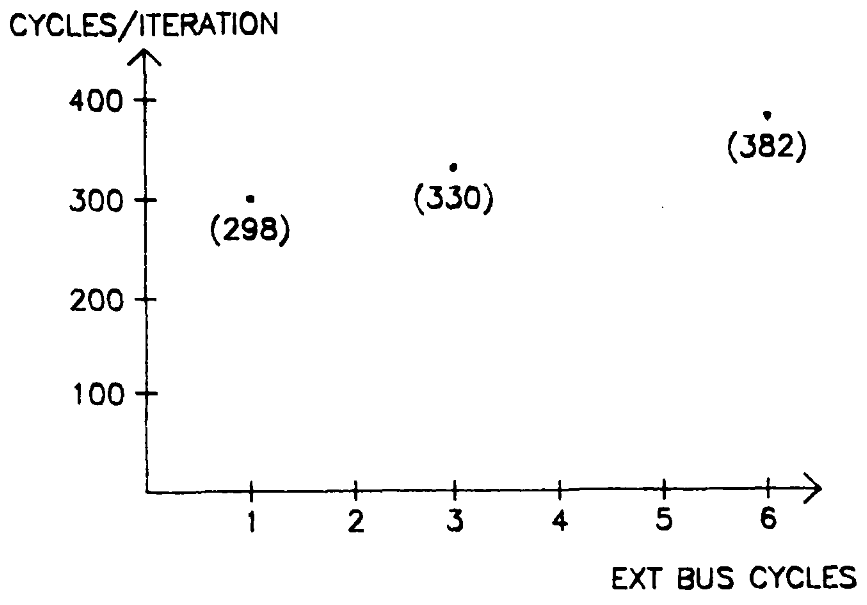
The 15 node network with long connections was used to measure the effects of the width of the external bus data path and the multiplier speed on the overall performance of the system. The multiplier speed was set to 20 clock cycles, and cases were run in which the external bus messages were divided into 1, 3 and 6 segments for transmission. Then the external bus was fixed at three segments per messages, and cases with multiplication requiring 5, 10, 20 and 40 clock cycles were executed. The results of these tests are presented in Fig. 17. The value for three segment messages and 20 clock cycle multiplication (i.e., 330 clock cycles per iteration) is used in the remainder of this theses as the basis of performance discussions.

For the other tests, midrange values for both of these parameters, i.e., three segment external bus transmission and 20 clock cycles for multiplication, were used. The results of these tests are presented in Table 4.

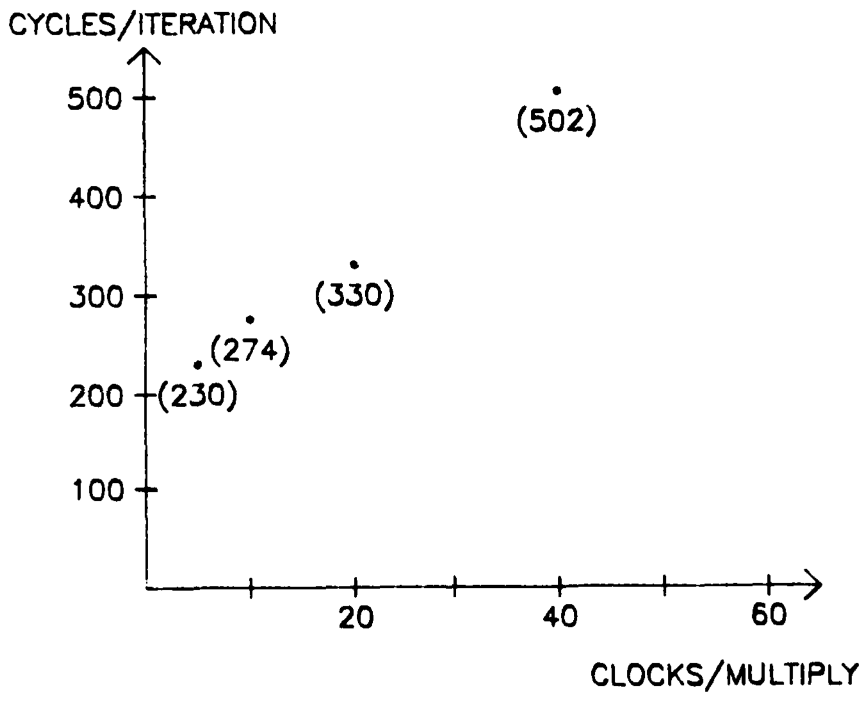
Table 4 - Performance for 16 and 36 Node Models

	16 nodes 3 color	16 nodes diagonal	36 nodes diagonal
Clocks/Iteration	210	214	222

These results show that once the iteration has started, the ordering schemes give virtually identical results for execution speed. The difference comes in the time needed to get the iteration started. The times shown above are for iteration at the last numbered node in the network. In the diagonal ordering, it takes some time after the iteration starts before this node is updated for the first time. In fact, the times required to complete the first iteration for



A



B

Figure 17 - Performance Results from Simulation

the three cases above were 129, 312, and 520 clock cycles respectively. Thus, the time to complete the first iteration was shorter than the average time for the three color ordering, since the use of assumed values reduced the amount of computation needed. But with the diagonal ordering, this factor was more than offset by the need for the first iteration to propagate through the network. As the larger case shows, this effect grows with the size of the problem. Thus, for large problems, use of the node color ordering would save a substantial amount of time in the operation of the system.

The small increase in time per iteration between the 16 node case and the 36 node case is caused by the use of a slightly different measure of performance for the 36 node case. The size of this problem caused it to take so much computation that even after 2.5 hours of DEC 20 CPU time, three iterations had not been completed. Thus, it was not possible to average the times for the second and third iteration at the last node, as for the 16 node problem. Instead, the times for the second and third iterations at another node were used. But since this node had different interconnections than node 16 in the other problem, the time for each iteration would be expected to vary. Ideally, several even larger cases should be run, to confirm that the time per iteration does not grow with the size of the problem as long as the interconnection pattern remains fixed. However, as mentioned above, it was not feasible to run these large test cases.

Due to the level of detail included in the simulation model, the results presented here should give a reasonably accurate representation of the level of performance obtainable from the ParSOR system. Obviously, the performance could not be expected to match exactly, since different operations are likely to require slightly more or fewer clock cycles in the final design. However, these highly detailed simulation results give a strong indication that an implementation of the ParSOR architecture could give performance on the order of the speeds presented here.

The simulation testing performed was confined to the operation of the system once iteration started, and included only the time to complete each iteration. It did not account for the times required to load the values of problem related parameters into the processing grid, make tests to determine when to stop iterating, or send the results back to the host. I will discuss these issues in Chapter 6 of this thesis.

Before going into these issues, I wanted to demonstrate that this architecture could be realized in hardware with the type of performance I assumed in the model. To do this, I constructed a hardware prototype of this architecture, using small scale and medium scale integrated circuits. The next chapter describes this prototype in detail.

CHAPTER FIVE - PROTOTYPE

It is quite easy, when designing systems on paper or writing simulation programs, to overlook significant details which will cause problems in implementation of the design. To avoid such oversights, and demonstrate the effectiveness of my design, I constructed a hardware prototype capable of demonstrating the operation of the main features of the processing element proposed in Chapter 3. This chapter describes the design, construction, and testing of the prototype, the software used to operate it, and the results of the testing performed on this hardware.

In the PARSOR architecture, the data paths and message lengths are large enough to permit fast operation on large problems. The system is designed for implementation in custom VLSI circuitry, which makes feasible the use of features such as registers and data paths 60 or more bits in width. However, such features are very costly in implementations using SSI/MSI chips. Construction of a prototype for an entire processing element would require about 1500 such ICs. Therefore, the prototype had to be simplified dramatically to make construction feasible. The prototype had to allow testing and demonstration of the basic concepts employed in the ParSOR architecture, yet it had to be implemented with fewer than 100 ICs in order to be feasible.

This was accomplished primarily by limiting the prototype to only two of the four CPs for one node, and eliminating the Arithmetic Unit. The use of two CPs allows modeling of all aspects of the communication among the CPs. In order to allow testing of different combinations of CPs, the circuitry was designed to be easily reconfigurable. The places in the circuit whose configuration depended on the position of the CP within the processor (up, down, left or right) were provided with jumper connections, so that the two CPs constructed could be changed simply to represent any pair of CPs in one PE, provided one was an external bus master and the other a slave. A block diagram of the prototype is shown in Fig. 18.

The arithmetic computations consist of simple addition and multiplication, so there is little to be learned by actual construction of this unit. Elimination of the AU allowed the data paths in the CP to be reduced to the size needed to carry the routing information (i.e., j, R and C). Since the values of the variables were not included in the messages, there is no need for registers to store these values. For example, the AU buffer was not needed at all, since it would never contain routing information. However, the status of this buffer was kept, and the processing of message traffic took place as if it was there. The routing information could be reduced from that required in the actual system due to the use of a single node. For example,

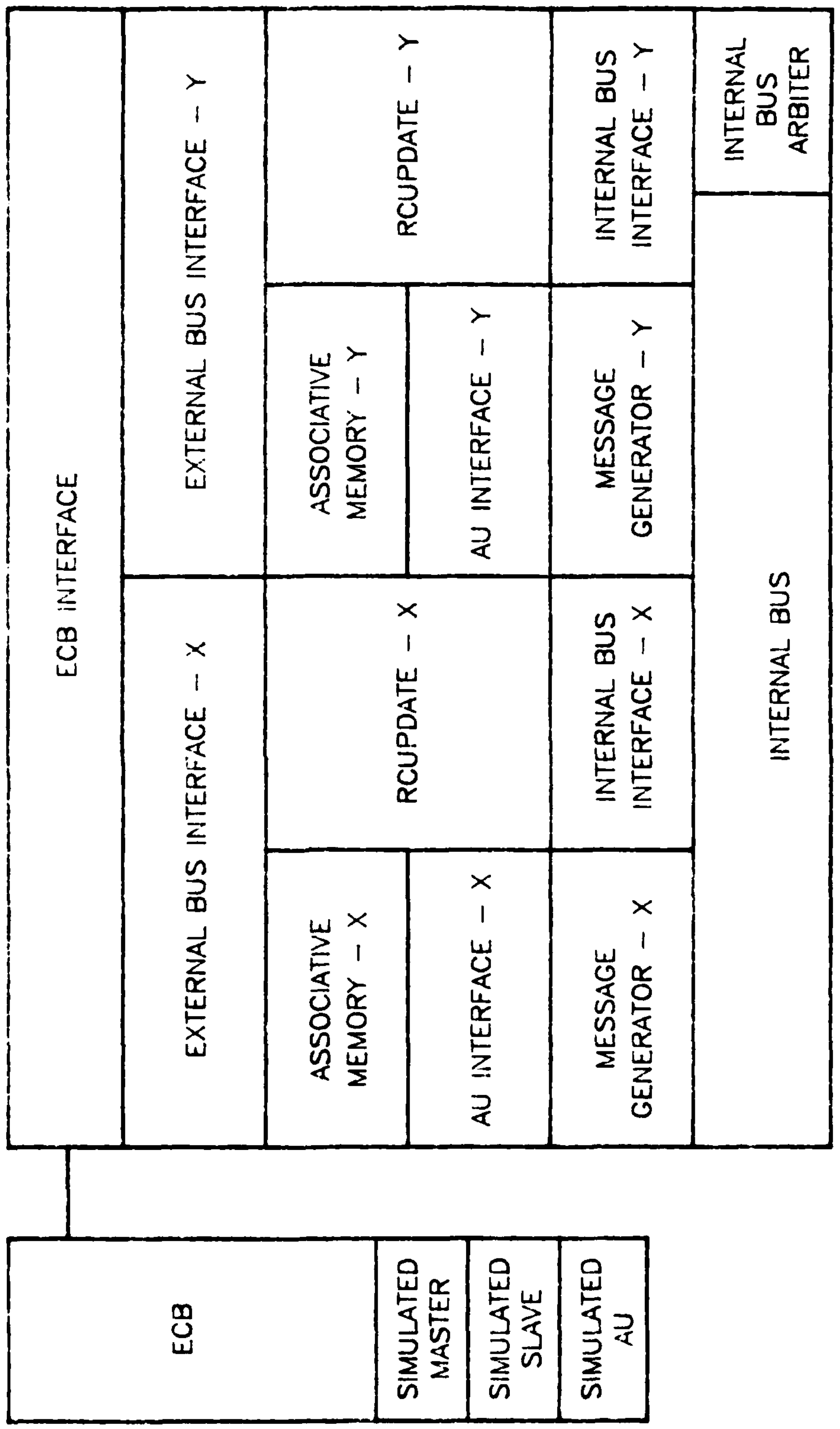


Figure 18 - Prototype Block Diagram

R and C could be reduced to two bits each, allowing them to take on the values -1, 0 and 1. The node identifier, j, could be represented in four bits, and still allow flexibility for testing. Thus, the width of the data paths was reduced from 60 bits to just 8. This allowed each data register to be implemented by a single chip, and was a convenient size for the application of many readily available ICs.

The scaled down size of the prototype and the messages allowed some simplifications in the working of the functional units. In the associative memory, there was no need to retrieve a matrix entry to correspond to the node number of the received message, since no actual multiplication would take place. Since the values of R and C were limited to -1, 0 and 1, there was no need for an adder in the RC update unit to update these values. The revised values could be determined directly from combinational logic. However, the prototype still needed to model accurately the time which would be required in the actual system for the working of the elements which were removed. Therefore, simple circuitry was added to model the timing of these units.

The operation of these CPs in a network of processors is modeled by simulating the other processors in software in a Motorola 68000 microprocessor. The processor used is on Motorola's Educational Computer Board (ECB), which consists of a Motorola 68000 microprocessor and supporting chips, with a simple operating system. The prototype includes interface circuitry to allow the required communications between the 68000 and the CPs. Several programs were written to allow the ECB to simulate different network activities, and facilitate observation of the functioning of the prototype under varying circumstances.

Our overall project includes two complete CPs, an Internal Bus Arbiter, and an interface to the ECB. It is constructed on a single circuit board, with dimensions of 8.5 by 11 inches. Extra sockets are provided for the use of jumpers to reconfigure each CP to represent any of the four possible CPs for one node. The project uses a total of 73 integrated circuits, excluding DIP switches, pull up resistors, etc.

Design Description

This section presents a general description of the design of the prototype, comparing it with the architecture of the proposed system as described in Chapter 3. Detailed descriptions of the circuitry used are presented later in this chapter.

As shown in Chapter 3, there are six functional units in each Communications Processor - external bus interface, associative memory, RC update, AU interface, message generator, and internal bus interface. Each of these units includes its own controller to maximize the parallelism in their operation. The simplifications introduced in the design of the prototype eliminated the need for two of these controllers - those for the associative memory and the RC update unit.

With the simplified design, both of these units were implemented in combinational logic. In the associative memory unit, the four bit index in the incoming message is compared with two four bit values stored in a single eight bit register. The NEEDED-HERE flag is set if the received value matches either stored value. No associated array value is retrieved. In the RC update unit, the revised values of R and C, the SEND-ON flag, and the destination CP number can be generated in only three SSI chips for all the possible CPs in a node. The inputs and outputs of this logic are connected to empty IC sockets, with connections wired in to both prototype CPs, so the appropriate values for a given CP can be provided by placement of jumper wires in the chip side of the sockets.

The only function left for the controllers to perform for these units is setting the flag indicating that the processing is complete. This task is accomplished in the prototype by cascades of D FFs. When the INFUL flag goes low, the FFs are cleared. When the INFUL flag is high, a one is clocked into the first of these FFs on each clock cycle. The output of one of the successive FFs is used to represent the AMDONE and RCDONE condition (since both of these units require the same length of time, a single signal can be used for both). This scheme allows different timings to be simulated with a simple wiring change.

The controller for the message generator requires the use of a counter, comparators, and address generation for the memory device which stores the message contents. This controller is implemented in discrete logic. A single chip "register file" is used to store the combinations of R and C for different outgoing messages. When the AU-IS-ANS flag is set by the AU interface, the controller generates the address for this register file, and also the other signals necessary to route outgoing messages to the NMBUF register. This controller uses a prestored count of the number of messages to send, and compares this value to a count it keeps of the number of messages it has sent. When it has sent enough messages, it signals the AU interface to clear the AUBUF.

The other three controllers required in each CP utilize Programmable Array Logic (PAL) chips, which allow the implementation of complex logic functions in a single chip. Each PAL can replace 5 to 10 SSI/MSI chips, so their use simplified the design and construction of the prototype significantly. The PALs must be programmed to perform the desired logic functions. This programming can be done in software, so it is easy to try different designs and modify them until the desired performance is achieved. However, the PALs available for this project were not erasable, so once the program is written to the chips themselves, it can not be changed.

The external bus interface unit requires a significant amount of status information from other units within the CP, as well as from the external bus control lines. Therefore, it required two PAL chips to implement this controller. These chips implement the communication function between CPs in different nodes over the external bus, and also maintain the correct status of registers within the CP, including setting and clearing the INFUL (input buffer full) and OBFUL (output buffer full) flags, and clearing the IBFUL (internal bus buffer full) and NMFUL (new message buffer full) flags when the data from these registers has been moved to the output buffer.

To simplify the interface to the ECB, the single bidirectional DATRDY line was replaced with two unidirectional lines - DATRDY_MS and DATRDY_SM. This enabled a reduction in logic for the discrete IC implementation. The single line is still preferable in the full scale implementation, since it reduces the number of pins required on a chip by four. In the prototype, the pin count was not a factor.

One significant feature was added to the architecture as a result of the protytpe construction. This had to do with keeping track of the status of the AUBUF and RTBUF registers. With the logic as originally designed, when an incoming message had to be sent to one of these registers and the appropriate buffer full flag was set, the INFUL flag would be cleared. However, this is not the correct action to take if the buffer full flag is still set to indicate the presence of a previous message. Thus, additional logic was needed to keep track of whether the value stored in the AUBUF and RTBUF represented the current value from INBUF, or old data. This function was implemented with the use of two D flip-flops and appropriate control signals. This is one example of the value of building a prototype of a system.

As described above, the AU interface did not require an actual AUBUF register for data storage. However, in order to model the operation of the system properly, it was necessary to implement the status keeping and communications functions of this unit. All these functions were implemented in a single PAL chip. This chip was programmed to maintain the status of the AUFUL and AU-IS-ANS flags, and communicate with the AU as simulated by the ECB computer.

The internal bus interface performed the communication functions between the two CPs which were built in the prototype. The controller for this unit was implemented by a single PAL chip. This chip generated all the status and control signals necessary to gain access to the internal bus and send or receive messages over that bus. In the actual system, this unit would have to monitor the internal bus address lines, to determine when a message was being sent to that unit. However, in the prototype, this would require separate PAL chips for each of the four possible CP addresses, even though only two would be in use at any time. Therefore, the address line decoding was moved to the internal bus controller, and decoded address lines were provided for all four CPs. As with the logic outputs in the RC update unit, these lines are connected by jumpers in an empty IC socket to the CPs, so that the CPs can be reconfigured easily. This allowed the same program to be used in the PAL chips for both internal bus interface units in the prototype.

The internal bus interface units rely on an internal bus controller to respond to their requests for access to the internal bus. In the prototype, this controller is implemented in discrete logic. Four D flip-flops are loaded with the pattern 0 0 0 1, and connected so this pattern will rotate continuously through them (i.e., as a ring counter). When the "1" moves into a location corresponding to a CP which has requested the bus, combinational logic asserts the IBGRANT signal to that CP, and prevents the bit pattern from circulating until the CP drops its request line. As mentioned above, this unit also serves to decode the address lines of the internal bus, so the logic within the CPs need not be changed to reconfigure them.

The circuit board for the prototype contains two CPs, the internal bus controller, and necessary circuitry to provide communication with the ECB which simulates the rest of the network. Data can be loaded into each CP (e.g., for the associative memory and the new messages to be generated) by DIP switches. Registers are provided for the storage of these values in the associated units. To load them, the switches are set for the value desired, and in the case of the register file in the message generator, the address to be used. Then a pushbutton switch is used to cause the value on the DIP switches to be stored in the register. A reset switch is used to send a reset signal to all the functional units which contain state information. Thus, the prototype can be set to many different configurations, and used to measure the performance of the communication processors under a wide variety of circumstances.

A few functions of the CP can be tested by applying externally generated status signals and clock pulses, and making measurements. However, for more complex tests, there are too many signals to apply manually. This was the reason for the use of the ECB computer. This computer generates all the necessary external signals and clock pulses to simulate the behavior of the network in which the CPs we constructed would operate. The interface to the ECB provides bidirectional data registers which can be used to send data to the prototype and receive data back from it. The software which operates the ECB must configure the I/O ports appropriately for the operation being performed. Then it follows a basic pattern of sending a clock pulse, reading the status and other signals received from the prototype, and asserting any new status signals.

Since the software controls the sending of clock pulses, the ECB can change any status and data lines in between clocks. Thus, the single ECB can appear to the prototype to perform several functions at the same time, such as send and receive messages over the external busses and the AU interface bus. This scheme allows the ECB to simulate the actions of several other processors in the network, as well as the AU for the node in which the CPs reside. However, the ECB can not simulate all these actions at full speed, since it must do a significant amount of processing for each clock cycle. Thus, the operation of the prototype must be measured in terms of numbers of clock cycles to complete certain operations, just as with the simulation. The operations taking place within the CP depend only on the clock pulses and the data values at the interfaces, so the prototype does in fact verify that these functions work correctly.

This scheme for operation of the prototype occasionally causes strange looking results in the tests. For example, the DATRDY signals generated by the prototype do not drop between segments when a CP is sending out a message to the ECB. This is because the software will control the reading of this status line on successive clock cycles and latch in the correct values. However, when a CP is receiving a message, it uses the rising edge of this signal to latch in data, so the ECB must assert this signal separately for each segment. Thus, the signals look somewhat different, even though they are intended to perform the same function. Also, in the actual system, the changes in inputs to a node are closely synchronized with the rising edges of the clock pulses. However, in the ECB, the software may change the value of an output flag at times far from clock edges. Thus, the response of the CP may not appear to be related to the clock in many situations. None of these concerns invalidates the results of the prototype testing, but they should be considered in examining the test data.

Prototype Testing

Several versions of the software for the ECB were written in order to test various functions of the system. This software was used to drive the prototype, and the operation of the hardware was observed by the use of a logic analyzer (HP 1630D). Appropriate sets of signals were selected for monitoring and recording for each type of test.

The recorded waveforms, along with descriptions of the test situation, are shown on the following pages.

The logic analyzer used allows only five characters for signal names. In most cases, the signal names shown on the waveform traces are the same as used in the design descriptions, or obvious contractions. Since there are two DATRDY lines for each external bus, a distinction is required. Also, it is necessary to distinguish between the CPs simulated in the ECB and those built in the prototype. Thus, each data ready signal is labelled DRx2y, where the "2" indicates "to" and x and y are replaced by M for the prototype master, S for slave, or E for a simulated CP in the ECB. For example, the signal used by the prototype master to indicate that it has data ready is labelled DRM2E. In other cases, a signal name is shortened and an M or S is added at the end, since the same signal in both CPs is shown on the same plot.

Other signals listed are:

CKRTB - latch data into RTBUF

EBST - state of external bus controller (3 bits)

SETIF - set INFUL flag

CLRIF - clear INFUL

IBGT - internal bus grant

AUDR - AU data ready

The following sections describe five tests which were run to verify the performance of the prototype under various conditions. In order to clarify what is happening in each test, the block diagram of the prototype has been reproduced for each, with arrows added to show where the signals are going.

In the first test, multiple messages were sent from the simulated master in the ECB to the prototype slave CP as rapidly as possible, as shown in Fig. 19. The messages contained values of R and C which indicated that they should be sent on to the master CP, and then out to a CP simulated by the ECB. The j values in these messages did not indicate that they were needed by the AU. The simulated CP which would receive data sent out by the master CP was not active in this run. This has the effect that the BUFFUL flag is never set, so the master continues sending data out whenever it has data to send. In this configuration, the master can send a message to the slave every 7 clock cycles, as shown by the waveform trace in Fig. 20.

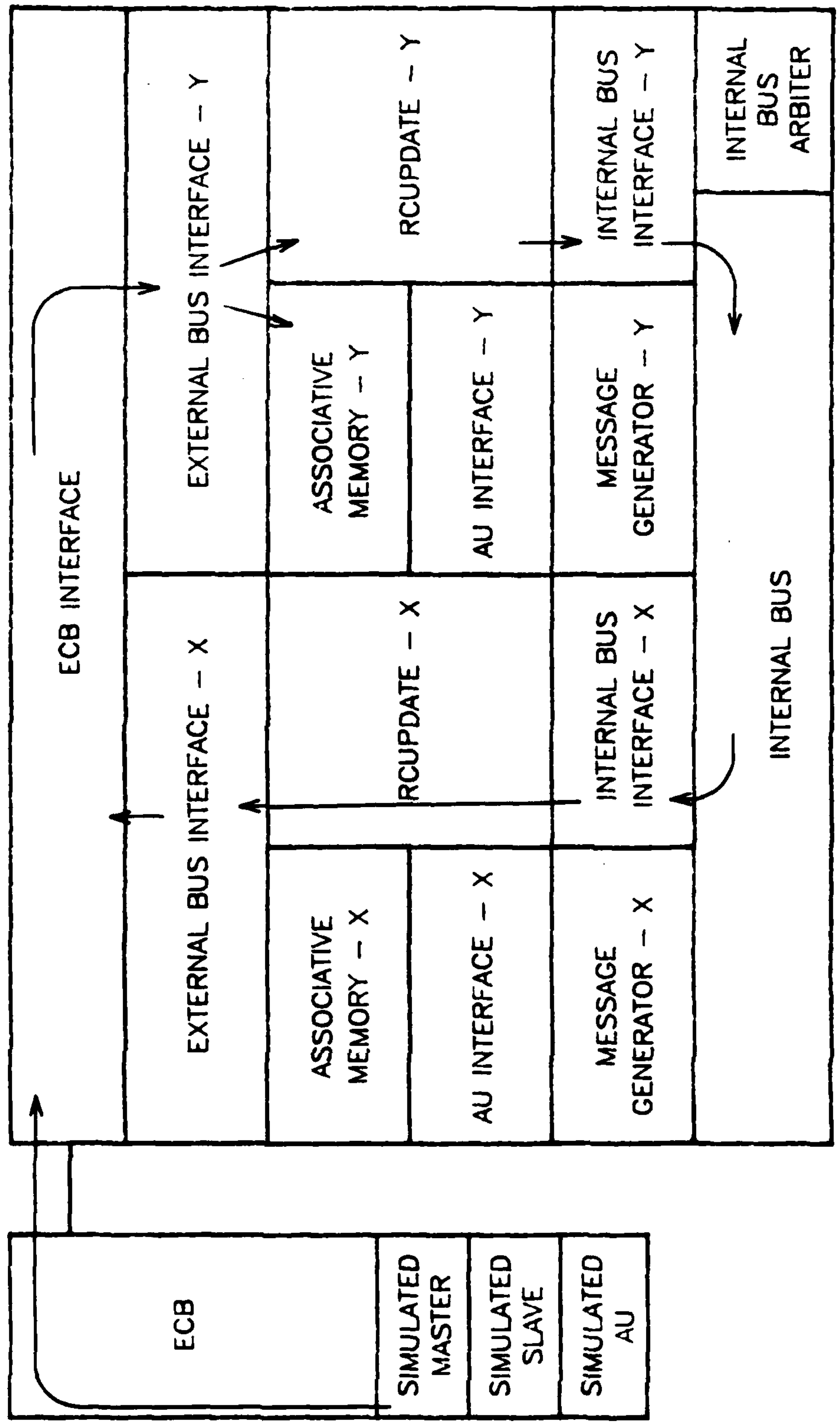
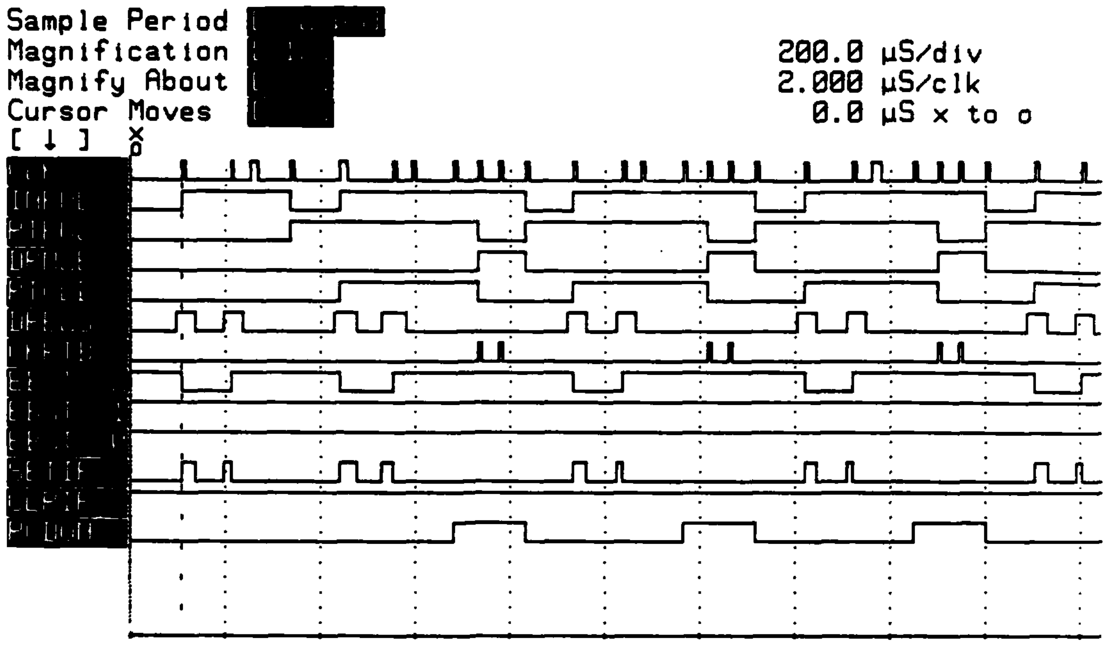


Figure 19 - Signal Flow, Prototype Test 1



Sample Period
Magnification
Magnify About
Cursor Moves

200.0 µS/div
2.000 µS/clk
0.0 µS x to o

Figure 20 - Prototype Test 1

The second test is similar to test 1, except multiple messages are sent from the ECB slave to the prototype master CP, as shown in Fig. 21. In this case it was necessary to program the ECB master to receive data from the slave, since otherwise, the GRANT signal would not be set. As with test 1, it was possible to send a message to the master every 7 clock cycles. This is shown by the waveforms in Fig. 22.

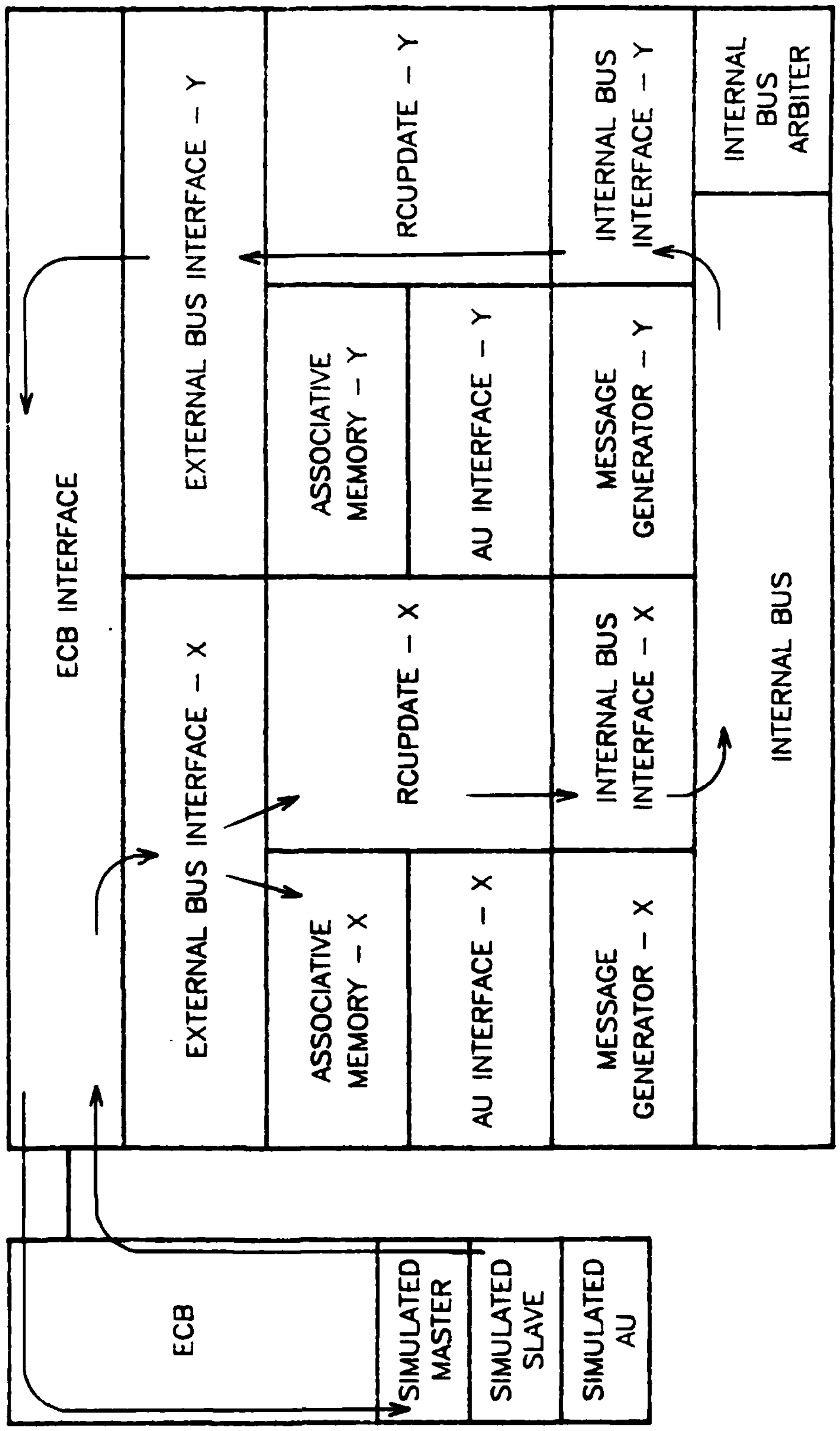
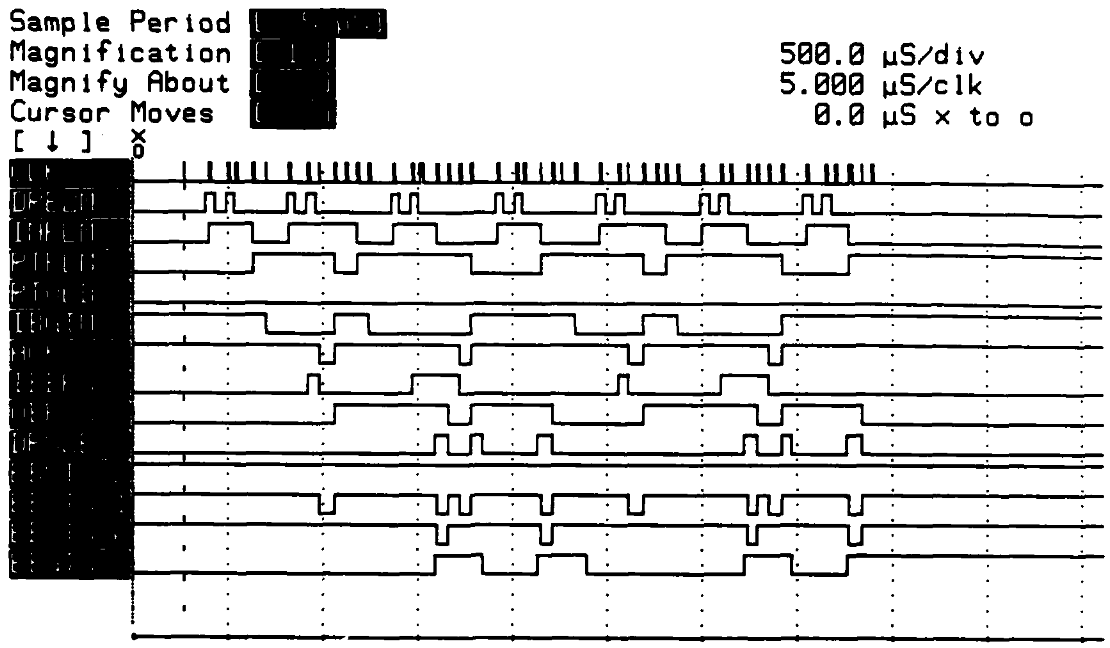


Figure 21 - Signal Flow, Prototype Test 2



Sample Period [illegible]
Magnification [illegible]
Magnify About [illegible]
Cursor Moves [illegible]
[↓]

500.0 µS/div
5.000 µS/clk
0.0 µS x to o

Figure 22 - Prototype Test 2

In the third test, messages are sent to both the master and slave at the same time. This is illustrated in Fig. 23. Both messages need to be sent on, but neither is needed at this node. This test verifies that the messages do not get lost or interfere with each other in the internal processing at the node. In this case, one of the messages takes 9 clock cycles to pass through this node, while the other takes 14, as shown in Fig. 24. Thus, the figure of 10 clock cycles derived in Chapter 3 for the time required for one node to pass a message is reasonably accurate.

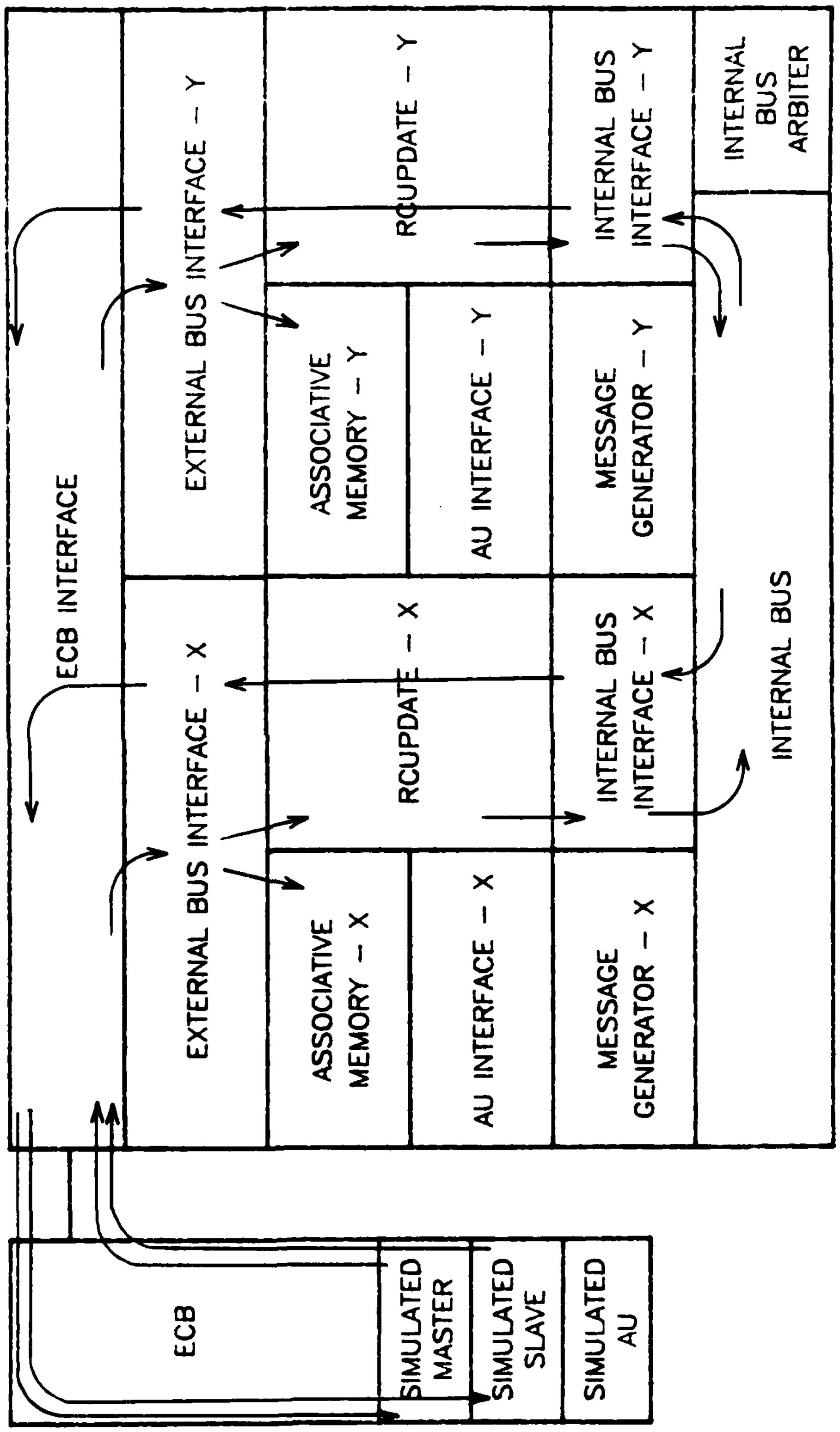
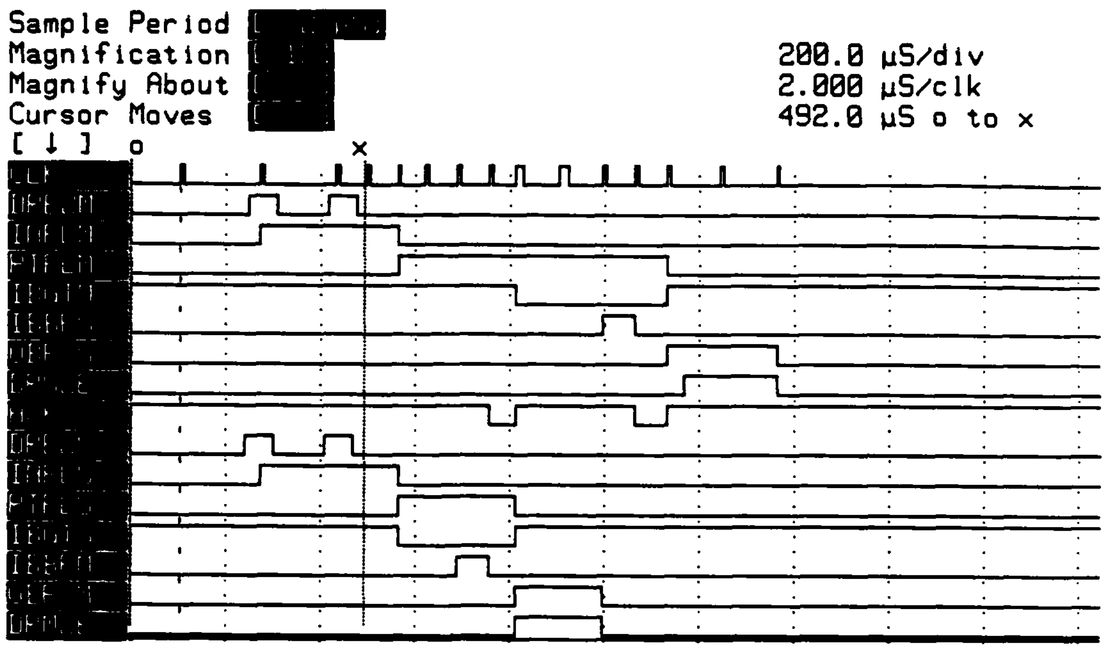


Figure 23 - Signal Flow, Prototype Test 3


Sample Period
Magnification
Magnify About
Cursor Moves

200.0 µS/div
2.000 µS/clk
492.0 µS o to x

Figure 24 - Prototype Test 3

In the fourth test, the AU has sent data to both CPs (prior to the time of the waveforms shown), so their message generators will send out new messages. Then messages are sent to both CPs simultaneously, which need to be sent on but are not needed here. This is shown in Fig. 25. Thus, the CPs must keep track of both the messages they are generating and the messages they are receiving and passing on. All these messages are processed accurately, and reach the proper destinations, as shown in Fig. 26.

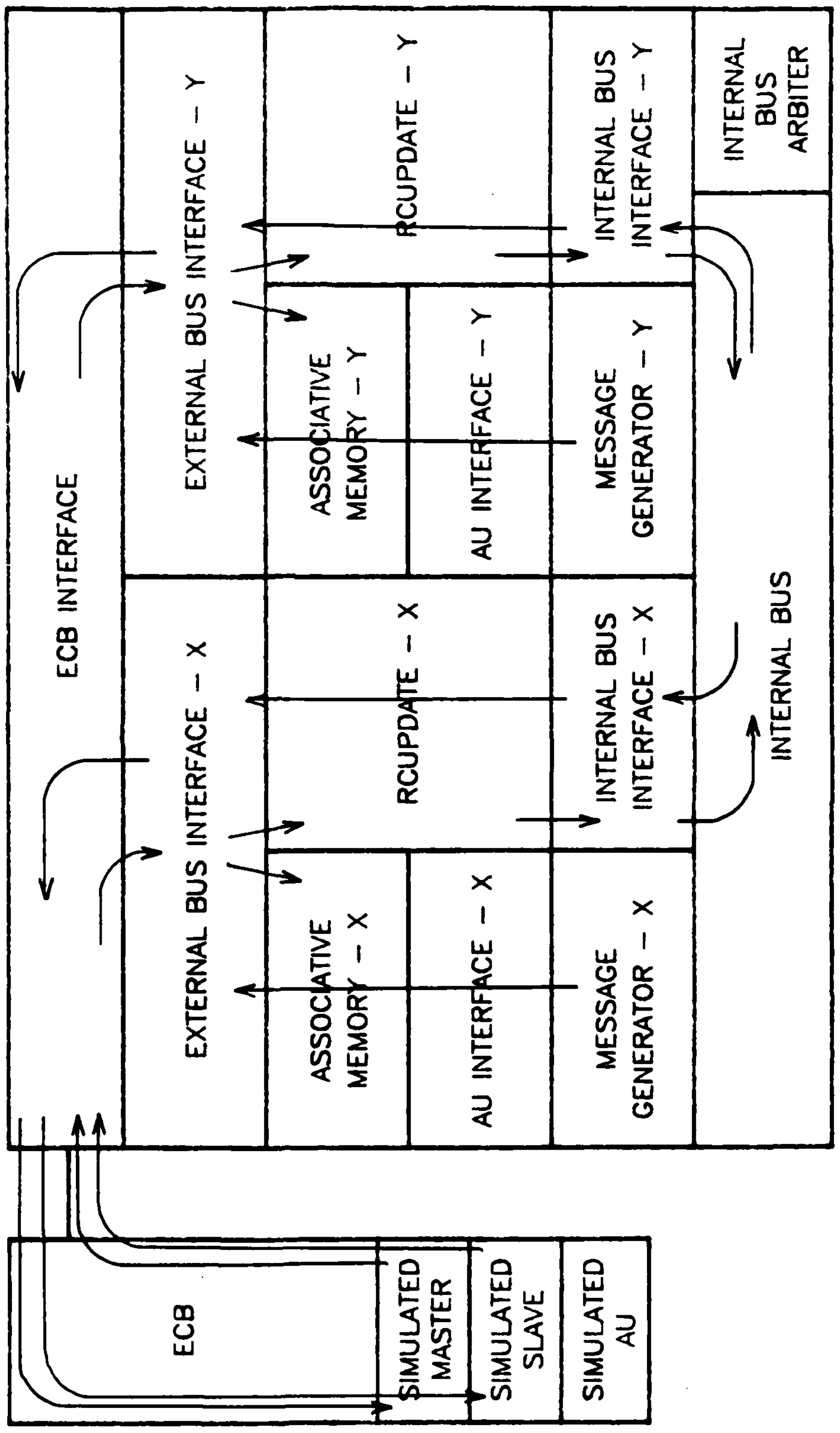
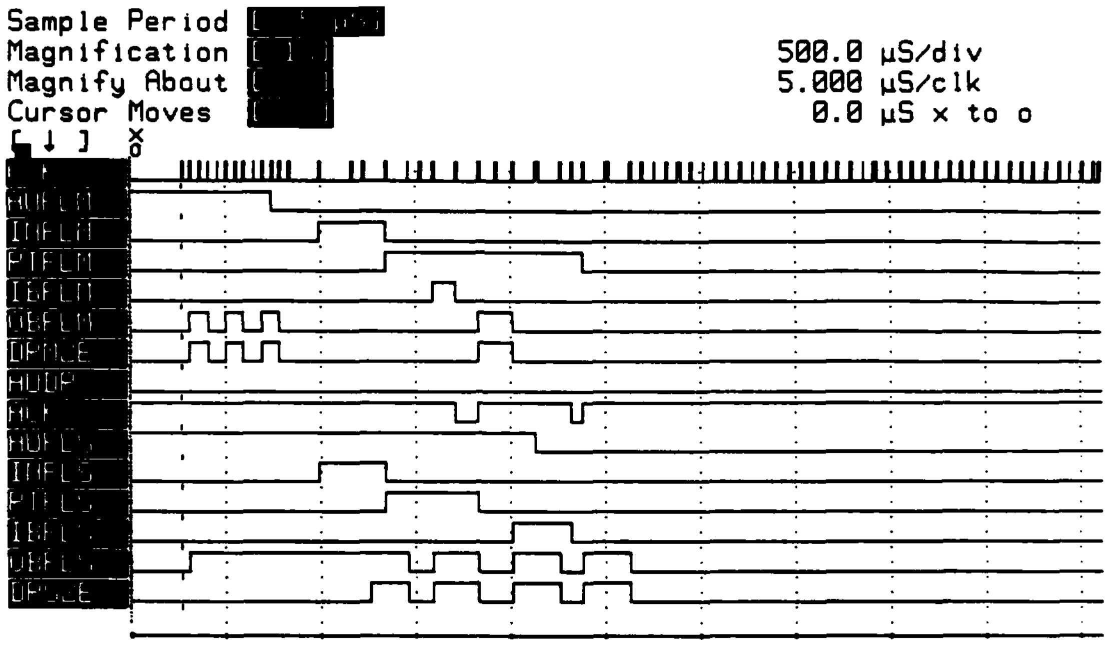


Figure 25 - Signal Flow, Prototype Test 4



```
Sample Period [     ]
Magnification [     ]                     500.0 µS/div
Magnify About [     ]                     5.000 µS/clk
Cursor Moves  [     ]                       0.0 µS x to o
```

Figure 26 - Prototype Test 4

In the last test, messages are sent simultaneously to both CPs, with j values which indicate the data is needed by their AU, and R and C values indicating that the messages must be passed on. The ECB is programmed to respond to the AU interface signals and read the data from these interfaces. The data flow for this test is shown in Fig. 27. Figure 28 shows that all the data is routed to the correct locations.

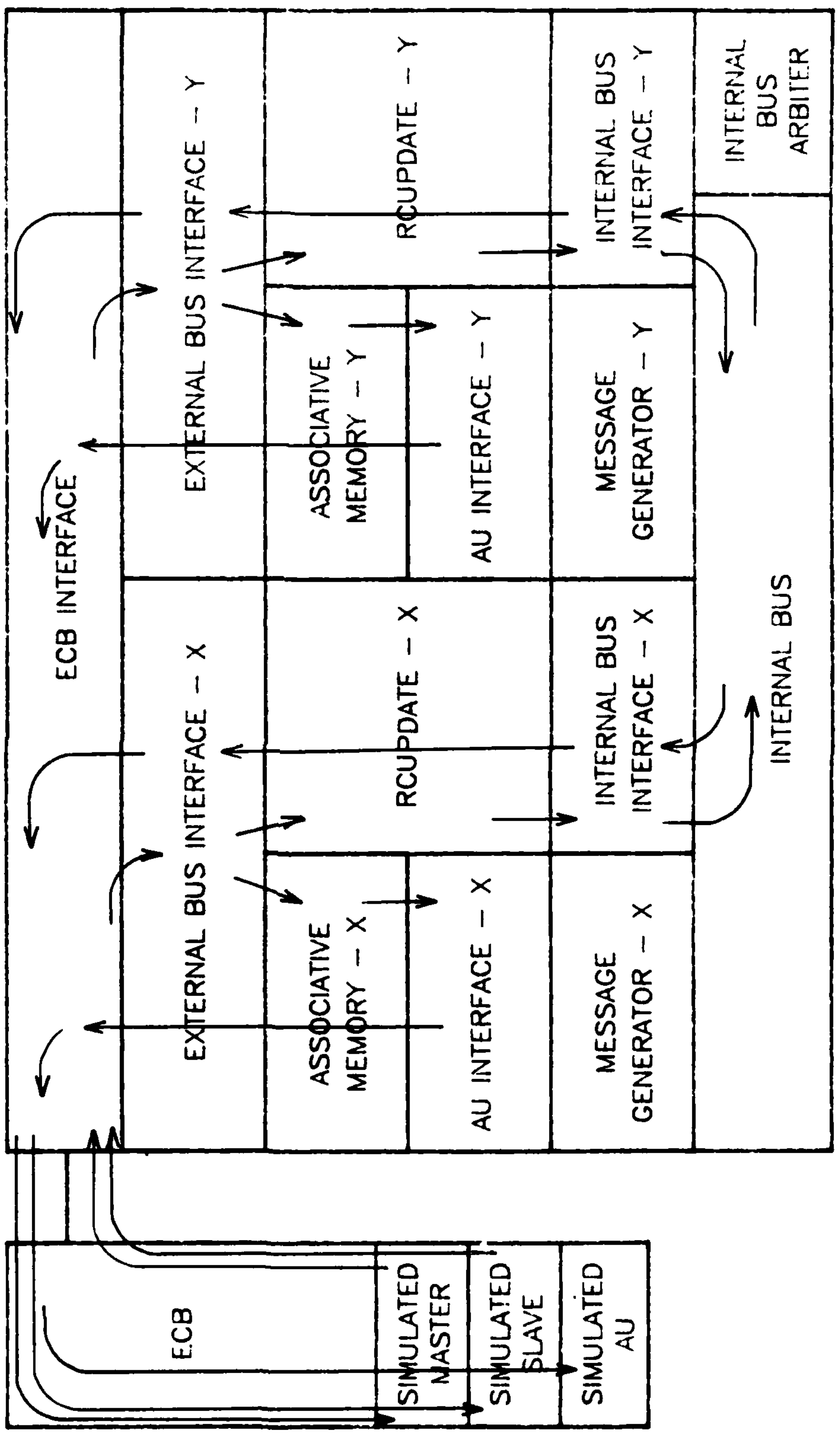
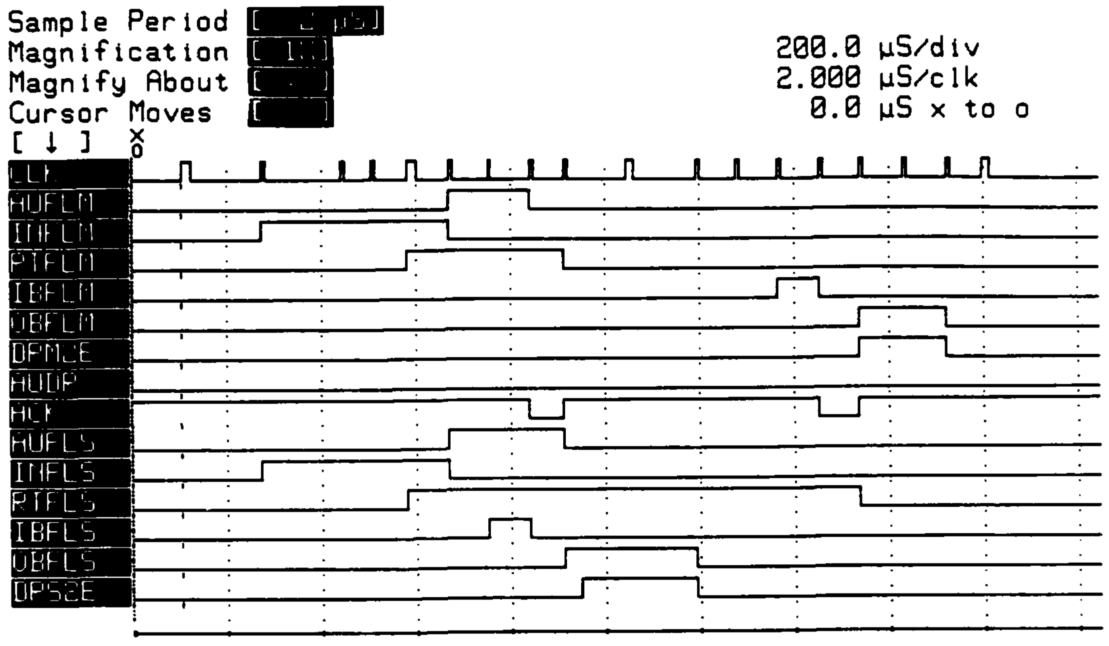


Figure 27 - Signal Flow, Prototype Test 5



Sample Period [2 µS]
Magnification [1:1]
Magnify About [ ]
Cursor Moves [ ]

200.0 µS/div
2.000 µS/clk
0.0 µS x to o

Figure 28 - Prototype Test 5

These tests on the prototype show that the proposed architecture is capable of handling all the types of messages which are required in this system, and pass them on quickly and with a minimum of interference. The times measured in these tests compare closely with those assumed in the simulation of the system as described in Chapter 4.

Design Details

This section provides circuit level descriptions of the design of the prototype hardware, and more detailed description of the software.

General Comments

Many of the drawings which follow show a clock signal, labelled CLK. This signal is generated by the ECB interface, and enters the prototype on one of the ribbon cable connectors which connect it to the ECB. If the ECB interface is not used, a manual clock can be applied to this pin. Several devices used in the construction of the prototype provide for an active low reset line. This was implemented with a DPDT switch and two 7404 inverters connected as a debouncer. This arrangement provides both active low and active high reset signals. We adopted a convention that any active low signal name would begin with an underscore character, e.g., the active low reset line is called _Rst, while the active high one is called Rst. This convention was particularly helpful when we were using the software package for programming the PALs. Unfortunately, the system used to generate the drawings (BRUNO, a locally developed software package for the HP-1000 computer) would not handle the underscore character. Therefore, in the drawings, a "/" is used for this purpose. The text will adopt this latter convention.

Several 7474 dual D FFs are shown in the drawings. Each of these is provided with an active low preset input. Unless otherwise noted, all of these inputs are tied to a logic one, since presetting is not desired. Logic "1" and "0" inputs are shown in several places in the drawings. The 0's simply involve tying the line to ground. The 1's are generated by the use of a pull-up resistor connected to the power supply.

The registers used in the prototype are 74LS374 chips. These chips provide 8 D flip-flops with a common clock line and tri-state outputs. The use of tri-state outputs allows several register outputs to drive a single set of inputs, provided only one of the outputs is enabled at any time. Thus, for example, both NMBUF and IBBUF outputs are connected to the inputs of OUTBUF. Separate output enable signals, labelled /SND-NMB and /SND-IBB, ensure that only one of these will be enabled at a time. In other applications, such as INBUF, the outputs can be enabled all the time. In these cases, the output enable line on the chip (active low) is simply connected to ground.

External Bus Interface

The external bus provides four data lines, while the messages to be sent are 8 bits long. Thus, each message must be sent in two parts. This requires that the controller be capable of selecting which bits to send, and keeping track of which bits it has already sent. The convention used is that the four most significant bits are sent first.

The signals used for communication between CPs over the external bus are shown here:

```
      |                  MASTER                  |
      |__________________________________________|
          | | | |     |        |      |      |      |
   Data   | | | | REQ| GRANT|   BUF|   DAT| DAT|
          | | | |     |        |   FUL|   RDY| RDY|
       ___|_|_|_|_____|________|______|    MS|  SM|__
      |                                          |
      |                  SLAVE                   |
```

Figure 29 - External Bus Control Lines

The sequence of operations for sending data in each direction are described below:

```
MASTER -> SLAVE

     Wait until BUFFUL is not asserted

  -> Place data on data lines
 |
 |   Assert DATRDY
 |
 |   Drop DATRDY (This step not included in prototype)
 |
  -- Repeat until entire token has been sent

     BUFFUL will be asserted by slave

SLAVE -> MASTER

     Assert REQ

     Wait for GRANT
```

```
-> Place data on data lines
|
|  Assert DATRDY
|
|  Drop DATRDY (Not included in prototype)
-- Repeat until entire token has been sent
   Master drops GRANT
   Drop REQ
```

Each processor should latch in the data from the data lines when the rising edge of DATRDY is sensed. As mentioned above, the prototype CPs do not execute the "Drop DATRDY" step for the first portion of data transmission, in order to be compatible with the operation of the ECB. However, they do expect the ECB to perform this step, and in fact use the DATRDY line as the clock input to the 73LS374 registers.

The circuitry to implement this unit is shown in Fig. 30. To receive the incoming messages in two portions, the inputs of the four least significant bits of INBUF are connected directly to the external bus data lines. The outputs of these four bits are connected to the other parts of the CP which need this data (R and C), and also to the inputs of the four most significant bits of INBUF. Thus, when data is sent over the external bus, the first four bits (the most significant bits of the message) will be latched into the least significant bits of INBUF on the first DATRDY pulse. On the second DATRDY pulse, these bits will be

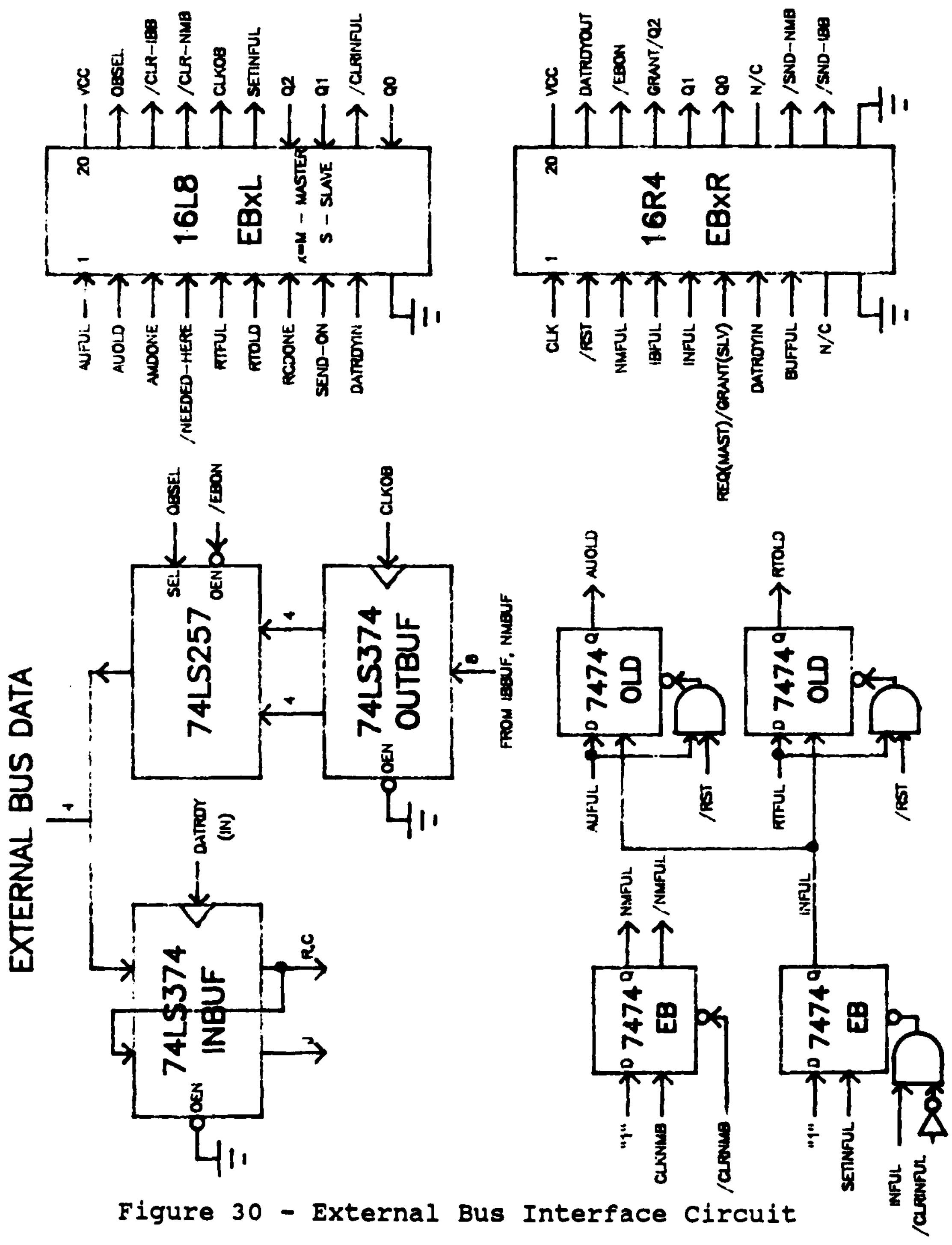


Figure 30 - External Bus Interface Circuit

latched into the most significant bit positions, and the rest of the message will be latched into the low order bit positions.

A 74LS257 multiplexer is used to select which bits to send. The inputs to this chip are wired to the outputs of OUTBUF. The OBSEL signal causes it to select either the most significant four bits or the least significant four bits for its outputs. The outputs of this device are tri-state, so the /EBON signal can be used to control when this device drives the external bus data lines.

Four 7474 D FFs (i.e., two chips) are used to contain the status signals NMFUL, INFUL, AUOLD and RTOLD. These chips were selected for this application because they have separate clock and clear lines for each flip-flop, which simplifies the logic needed to maintain the proper status. Note the extra inverter and NAND gate at the input of the INFUL flip-flop CLR input. These were added as a result of testing, which showed that the /CLR-INFUL signal was not logically correct. This could be fixed by changing the logic in the PAL chip, thus eliminating these extra gates. We placed the gates in the circuit to verify the functioning of the circuit with the modified logic, rather than burn another PAL, then simply left them in place when the revised circuit functioned correctly. Thus, there is room for improvement in the construction of the prototype, in this and some other areas. This report documents the prototype in its actual configuration, without these improvements.

The major portion of the external bus interface is the controller. This unit is constructed from two PAL chips. The 16R4 chip contains 4 internal D flip-flops, and implements a state machine to control the operation of the external bus and the interface. The 16L8 provides additional combinational logic to implement many of the control signals required in the system. The configuration is as follows:

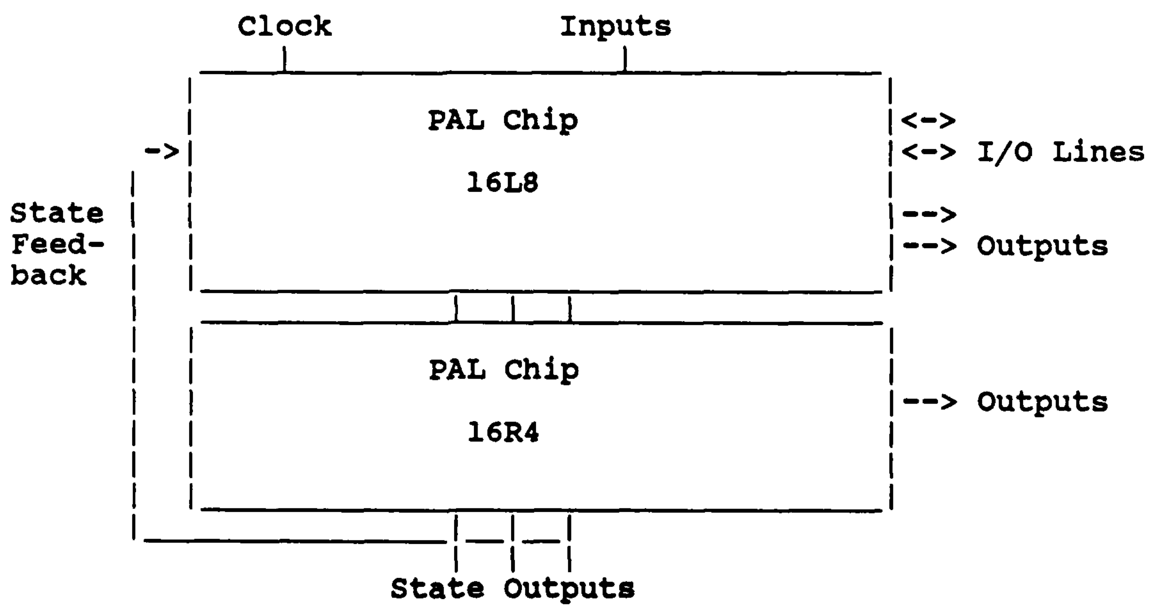


Figure 31 - External Bus Interface Controller

The details of these controllers depend on whether the CP is a master or a slave on the external bus. The PALs were programmed using Data I/O's ABEL (Advanced Boolean Expression Language) program. This program allows the designer to specify the functions desired for a chip in several different ways. For this application, the most convenient way of expressing the functions of the 16R4 chip was as a state diagram. For the 16L8 chip, simple logic equations were used. The ABEL program translates these inputs to a reduced logic fuse map for use by the PAL programmer. We designated the 16L8 chips EBxL (x replaced by M for master or S for slave), indicating External Bus Logic, and the 16R4s EBxR, for External Bus Registers. The inputs to the ABEL program, specifying the details of the operation of these chips, are shown in Appendix D.

Associative Memory

The circuit for this function is shown in Fig. 32. The DIP switches and pull-up resistors are used to generate logic ones and zeros for storage in the 74LS374 register. This register is capable of storing two four bit index values which are needed at this node. The values of the data set by the switches are latched in to the register at the rising edge of the _RST signal. This was intended to be expanded for testing of automatic data loading schemes, but these have not been implemented. The system could be used without the register, by using the values taken directly from the DIP switches as inputs to the logic.

The values stored in the register halves are compared with the value of j in the received message by the 7486 XOR

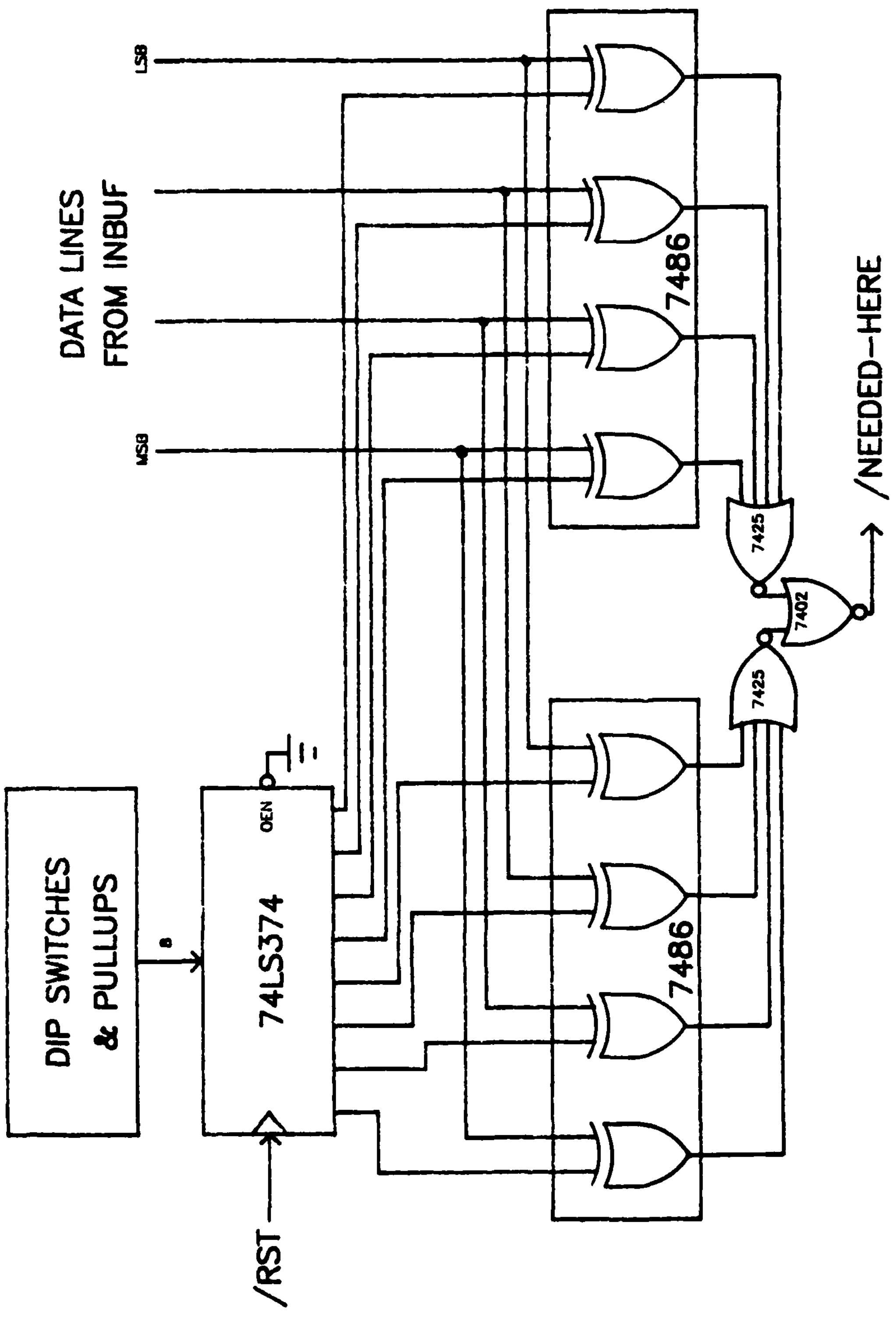


Figure 32 - Associative Memory Circuit

gates, and the 7425 four input NOR gates. The 7425 provides for a strobe input, which was not needed in this application, so we connected it to a logic "1" signal. If the bit pattern of the received j matches either stored value, a logic "1" will appear at the output of the corresponding 7425. A logical OR of these lines indicates that the data is needed by the AU of this processor. A spare 7402 NOR gate was available in another part of the circuit, so we used this gate, thus obtaining an active low signal, /NEEDED-HERE. This signal is used as input by several PAL chips, so it was easy to program them to accept it as active low, rather than active high. This saved on IC in the design.

RC Update Unit

As mentioned above, this unit in the prototype is considerably simpler than in the actual architecture. The combination of the data routing scheme implemented and the restriction of R and C to the range -1 to 1 allows all the functions to be implemented with a small amount of combinational logic. For example, if the received value of R or C is 0, the output is also 0. If only one of these is nonzero, it will be updated to zero (-1 + 1 or 1 - 1). Thus, only the case of both R and C nonzero results in a non-zero R' or C'.

The logic to be implemented is different for each CP (up, down, left and right). However, all the functions for all 4 CPs could be implemented with three ICs - one each of 7404, 7408 and 7432. Thus, it was convenient to collect all these logic functions in a single location, as shown in Fig. 33. This required a way to connect the inputs and outputs corresponding to a particular CPs to the rest of the circuitry for the CP. This is the purpose of the IC sockets labelled "Jumper Sockets" in Fig. 33. To configure a CP to represent a particular CP within a node, the user simply places jumpers on jumper sockets 1 through 6 connecting R, C, R', C', DEST, and SEND-ON to the values for the correct CP number (0-Up, 1-Down, 2-Left or 3-Right). Thus, there is only one circuit as illustrated in Fig. 33 in the prototype.

Arithmetic Unit Interface

As mentioned above, the data lines between the CP and the AU are not implemented in the prototype. Thus, this unit generates only the control and status signals. The control signals for communicating between the CP and AU are as follows:

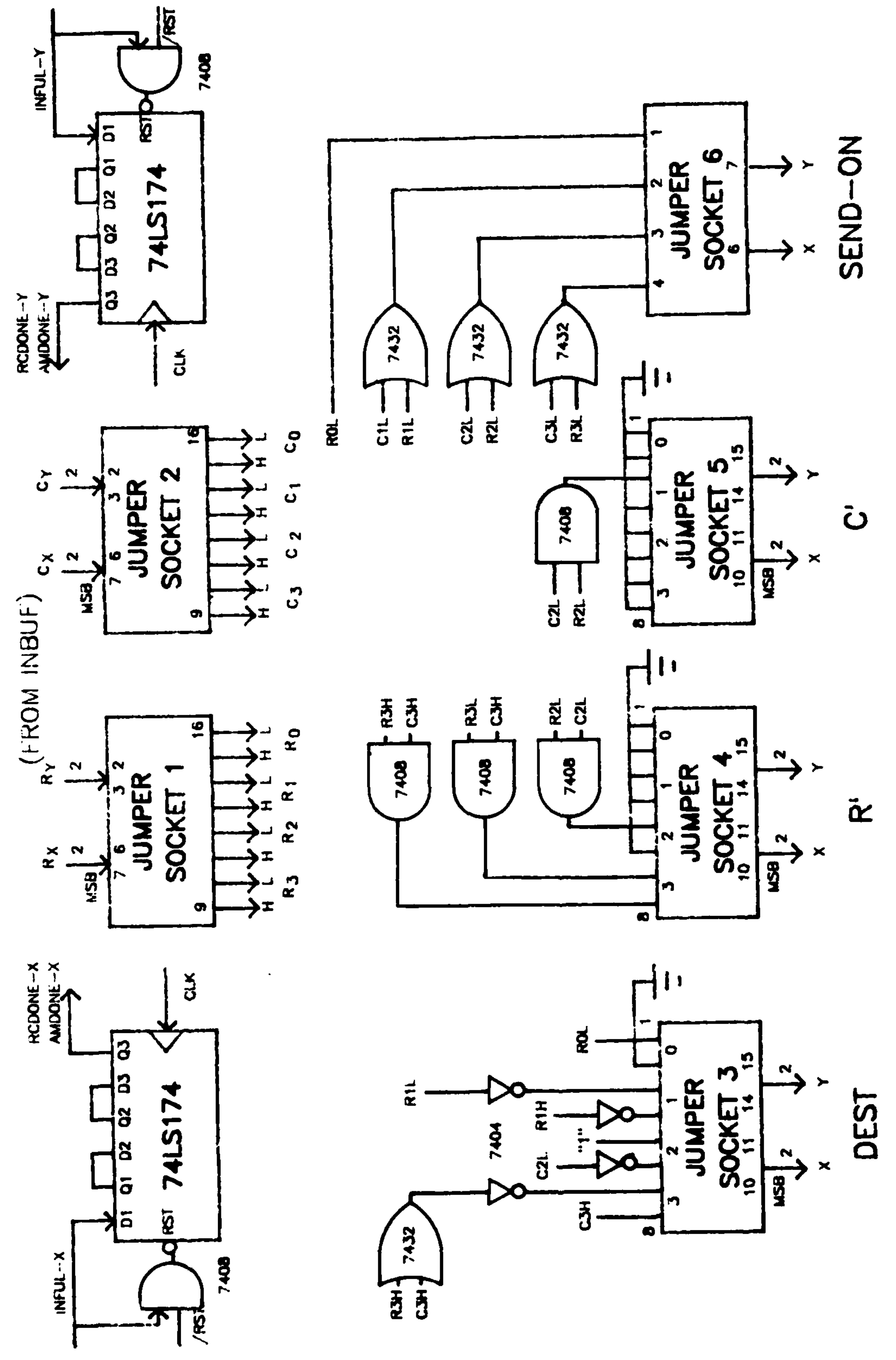


Figure 33 - RC Update Circuit

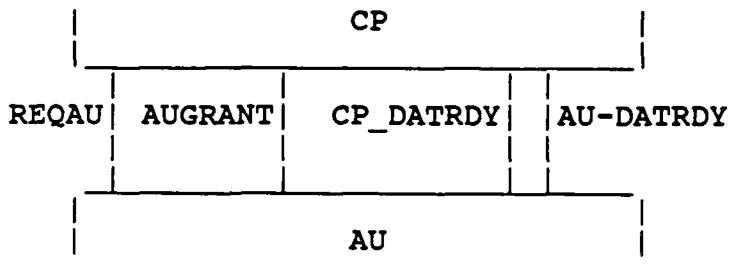


Figure 34 - AU Interface Control Signals

The timing of the control signals is as follows:

CP -> AU

CP asserts REQAU

Wait for AUGRANT

(Place data on data lines - not present in prototype)

Assert CP-DATRDY

Remove CP-DATRDY and REQAU

AU -> CP

(AU places data on the data lines - not in prototype)

AU asserts AU-DATRDY

AU drops AU-DATRDY

In addition to these control signals, the controller must maintain the status of the AUFUL and AU-IS-ANS signals. All of these functions are implemented in a 16R4 PAL, as shown in Figure 35. The logic used in this chip is described by the ABEL inputs listed in Appendix D.

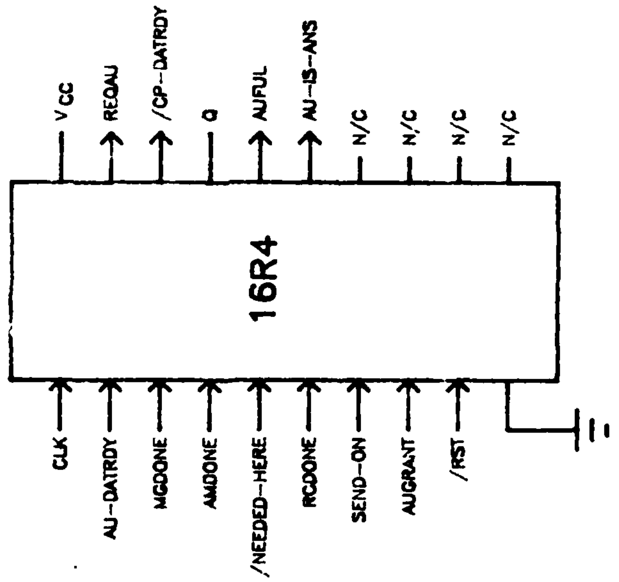
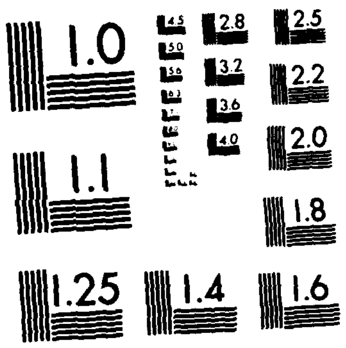


Figure 35 - AU Interface Circuit

AD-A171 256   A PARALLEL PROCESSOR FOR THE SOLUTION OF LARGE SPARSE   3/4
SYMMETRIC LINEAR SYSTEMS(U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH   D A ARPIN11986
UNCLASSIFIED   AFIT/CI/NR-86-124D   F/G 9/2   NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

Internal Bus Interface

The circuitry for this unit is shown in Figure 36. Note that registers IBBUF, RTBUF, and DEST all require tri-state outputs, which are enabled by control signals. The outputs of IBBUF are enabled when the external bus interface is ready to move the message contained there to OUTBUF. The outputs of RTBUF and DEST are enabled when this unit is sending data over the internal bus.

Two 7474 D flip-flops (one chip) are used to maintain the status signals IBFUL and RTFUL. This chip provides separate clock and clear lines for each flip-flop, thus simplifying the control of these signals. In testing the prototype, it was discovered that the timing of the CLKRTB signal did not always result in setting the RTFUL flag, due to a conflict with the /CLR-RTFUL signal. Therefore, the 7408 AND gate was added to assure that the RTFUL signal would be set within one clock cycle after the value was transferred to the RTBUF. This was adequate to insure proper operation.

The sequences of operations for sending and receiving data are as follows:

SENDING -

Assert REQIB

Wait for /IBGRANT

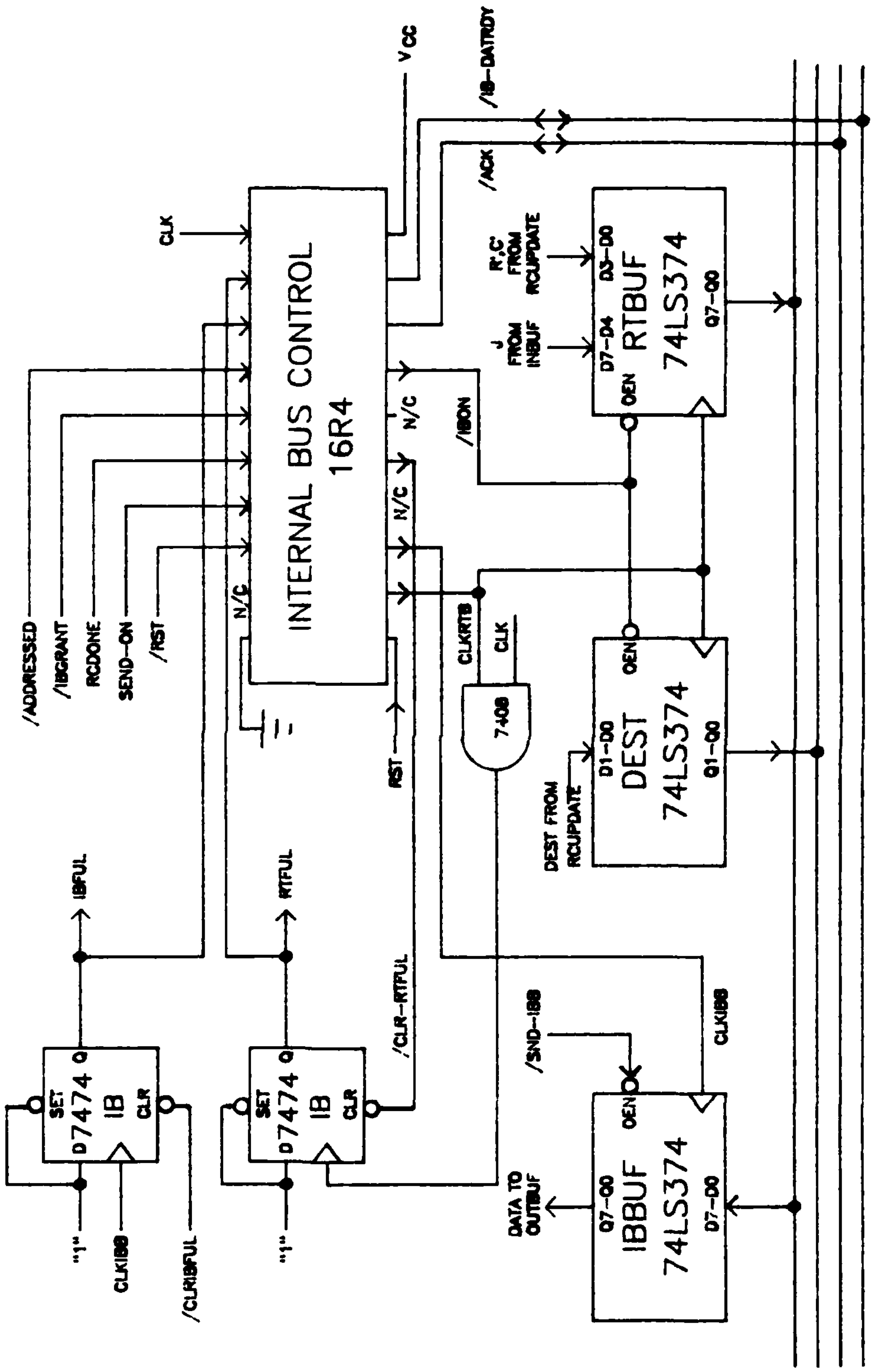


Figure 36 - Internal Bus Interface Circuit

Place data, destination address on Data and Addr lines

Assert /IB-DATRDY

Wait for /ACK to be asserted by receiver

Drop REQIB, /IB-DATRDY, Data, Addr lines

RECEIVING -

Watch for /ADDRESSED and /IB-DATRDY

Wait for IBFUL signal to be cleared

Latch in data from Data lines and assert /ACK

Drop /ACK

The controller which implements these functions is contained in a 16R4 PAL chip. The operation of this chip is described by the ABEL program inputs listed in Appendix D.

Message Generator

Figure 37 illustrates the circuitry which implements this function. The 74LS174 stores the index value for this node, which will become the j value in the outgoing messages. The unit must be able to generate a specified number of messages, each with different values of R and C. Two bits of the '174 are used to store the number of messages to be sent. The values of R and C for each message are stored in the 74LS170 register file, which contains four addressable storage locations of four bits each. DIP switches provide a way of specifying the write address and

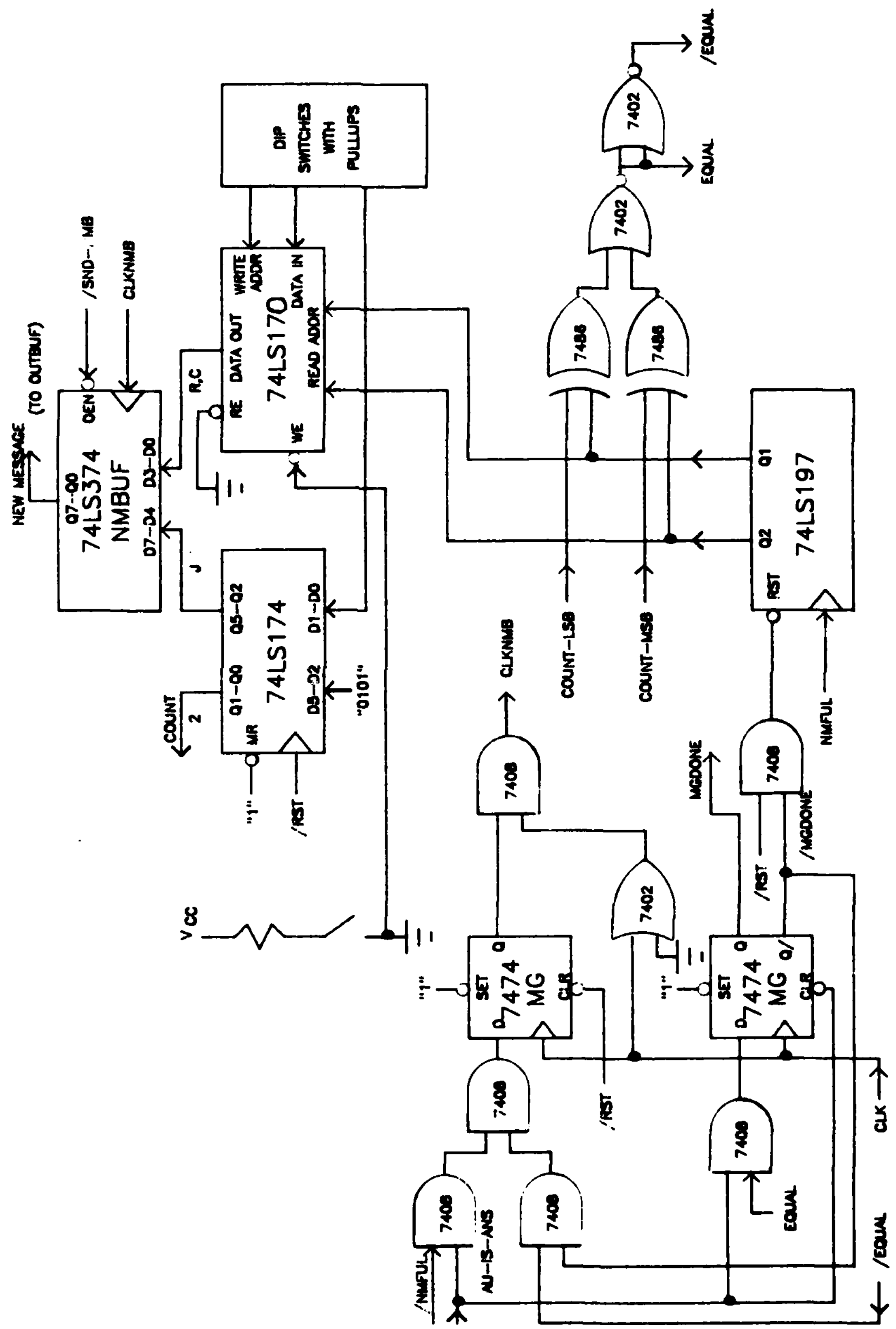


Figure 37 - Message Generator Circuit

data to be stored in this device. A separate pushbutton switch with a pull-up resistor provides the necessary clock pulse for storing data.

The combinational logic and two D flip-flops serve to detect when messages should be generated, and to control the sending of the messages so that a new message is placed in NMBUF only when it is ready to receive one. A 74LS197 counter keeps track of how many such messages have been generated. The two 7486 and two 7402 gates compare this count with the prestored number of messages to be sent. This comparison is fed back to the control logic, causing the operation to stop after the correct number of messages has been sent. With two bits for the count, up to three messages may be sent. When message generation is complete, the logic generates the MGDONE signal, telling the AU interface that the data in AUBUF is no longer needed, and resets the counter for the next operation.

Internal Bus Arbiter

The circuit diagram for this unit is shown in Figure 38. The heart of this unit is the 74LS195 shift register. This device is connected as a ring counter. When the /RST line is low, the binary number 0001 is loaded into this four bit register. When this line is high, clock pulses cause the bit pattern to circulate through the

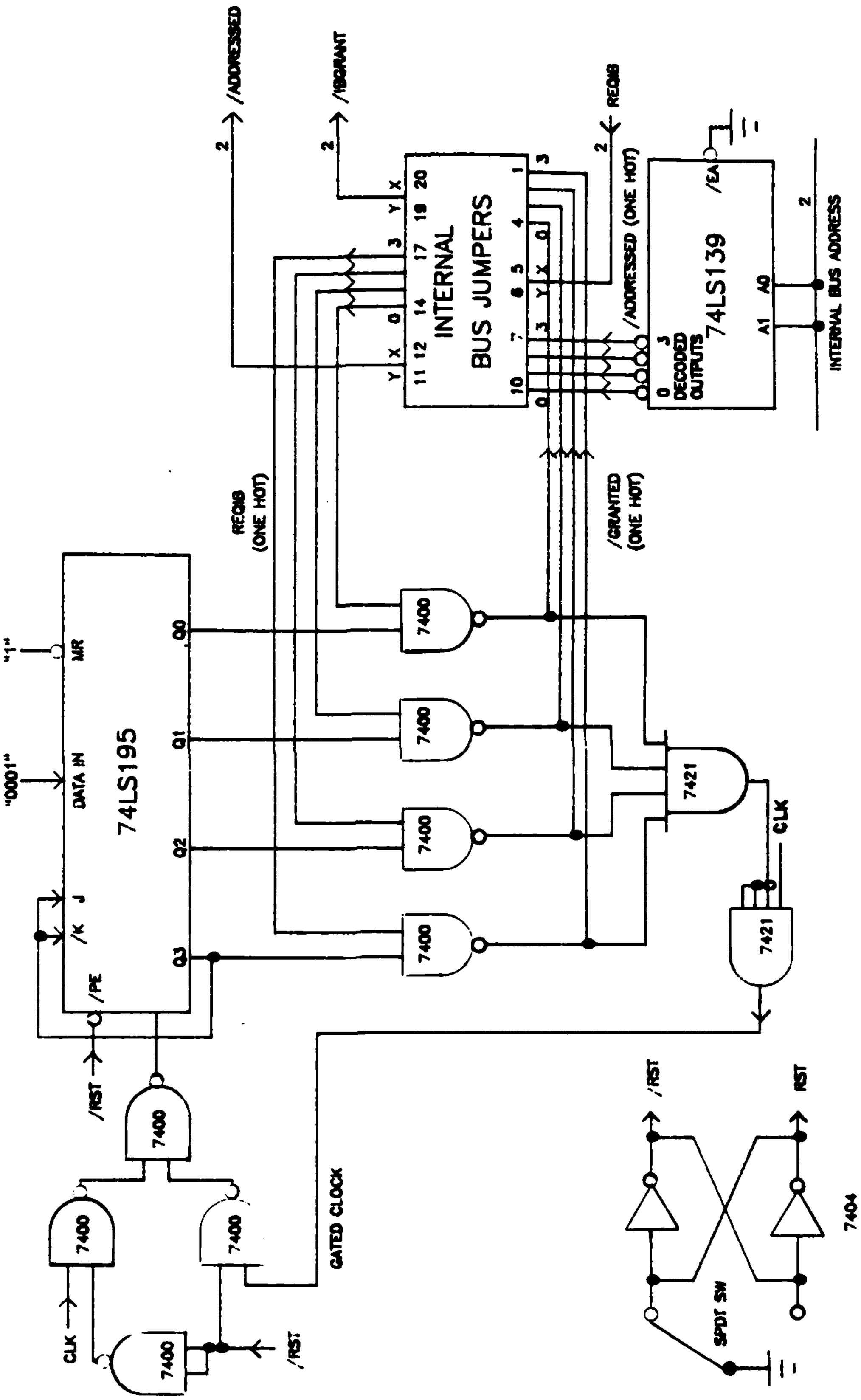


Figure 38 - Internal Bus Arbiter Circuit

register, so three bits are zeroes, and the other is a one. The outputs of this register are gated on a bit for bit basis with the REQIB lines from the CPs. When the "1" moves to a position corresponding to a CP which is asserting REQIB, the output of the corresponding 7400 NAND gate will go to zero. This signal comprises /IBGRANT for the associated CP. It also causes the output of the first 7421 AND gate to go low, thus preventing the clock pulses from reaching the '195. When the CP which was granted access to the bus is finished, it drops REQIB, which causes the NAND gate output to rise, and allows the clock pulses to reach the '195 again.

The clock signal must be present at the '195 in order to load the data when _RST is low. Four 7400 gates provide a selector, so that the clock pulses always get through when the _RST signal is asserted. The circuit which generates the _RST signal is also shown in Fig. 38.

As described above, the decoding of the addresses used on the internal bus has been moved from the individual CPs to the internal bus controller in the prototype, to allow for easier reconfiguration. This decoding is done by a 74LS139 decoder. As with the RC update unit, the outputs of this device are connected to the CPs by jumpers, so that the CP can be reconfigured to represent a different location within a node by a simple change of this jumper.

ECB Interface

To simulate the rest of the network, and the arithmetic unit for the node in which the prototype CPs are located, a Motorola Educational Computer Board is used. Several of the lines used to communicate with the ECB are bidirectional in nature, as described above. This interface implements these lines and all the data and control signals to the CPs.

Included in the ECB is a Parallel Interface/Timer chip (Motorola 68230), which includes several bidirectional data ports. However, one of these ports is buffered with single direction (output only) buffers on the ECB. Thus, more interface registers are needed than this device can provide. These additional ports were implemented by an Intel 8255A Peripheral Interface Adapter chip, which performs many of the same functions, but is less complicated to program. Use of this chip required address decoding and the generation of other timing and control signals, to enable the 68000 to send and receive data to it. The circuit which implements this interface is shown in Figure 39.

The I/O ports of the 8255A were memory mapped into the address space of the 68000 using the 68000 bus signals available via the J17 connector on the ECB. The 4 MHz clock signal had to be run through a Schmitt trigger inverter to eliminate excessive noise and distortion. The E5 address decode signal (range 70000 - 7FFFF hex) was used along with

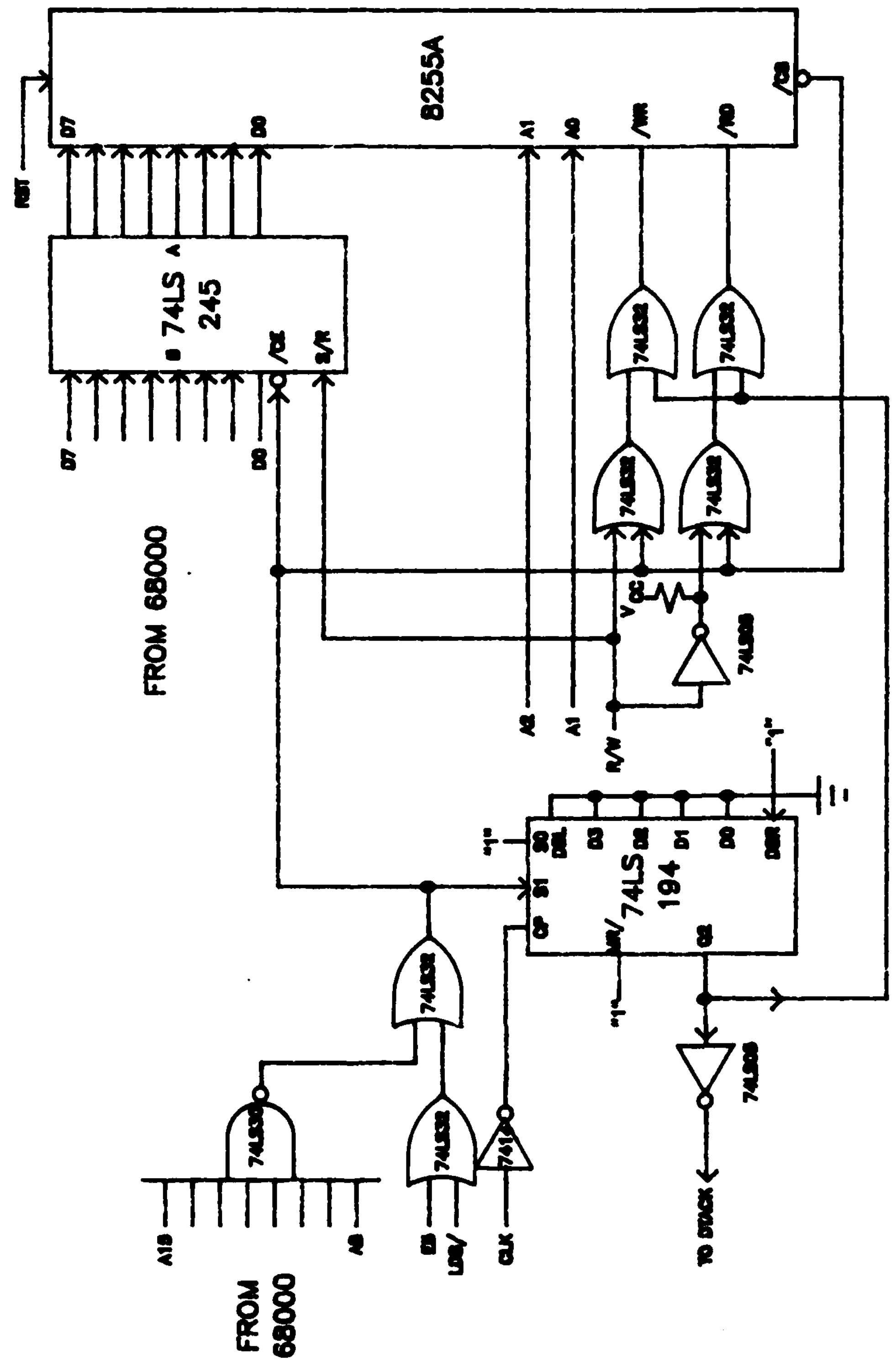


Figure 39 - ECB Interface Circuit

additional address decoding to narrow the valid address range to 7FF00 - 7FFFF hex. Since the 8255 is only an 8-bit device, only the lower half of the data bus (D0 - D7) was connected and the address decode signal was gated with the LDS/ (here, a "/" at the end of a signal name is used to indicate an active low signal, to be consistent with Intel documentation) signal to ensure that only byte read/write cycles to odd addresses would be valid. The DTACK/ signal back to the 68000 was provided by enabling a shift register to begin shifting in "ones" when a valid address decode signal was present. The shift register is cleared when the address decode signal is negated. This provides a flexible way of altering the timing of the DTACK/ signal to suit the requirements of the circuit.

Timing problems were encountered when trying to provide suitable RD/, WR/ and CS/ control signals to the 8255. The address decode signal was connected to the CS/ input, but because of the 68000 timing protocol, this signal was asserted after active RD/ or WR/ signals could be provided. Because the falling edge of RD/ or WR/ would occur before the falling edge of CS/, the 8255 would not recognize a valid command. In order to solve this problem, the RD/ and WR/ signals were gated with the CS/ signal so that they would be asserted after CS/. Now, however, the rising edges of the RD/ and WR/ signals were also after the rising edge of CS/ and the 8255 would not recognize the end of the commands and would not latch input data. Finally the RD/ and WR/ signals were gated with the inverse of the DTACK/ signal in addition to the CS/ signal and this arrangement proved to be successful.

Figure 40 illustrates the signals which connect the ECB to the prototype CPs. Some of these signal names begin with an M or S, denoting to which of the two CPs simulated in the ECB that signal pertains.

Appendix E contains a listing of a representative version of the software used to run the prototype. Comments in the listing describe the various functions performed by different parts of this program.

Conclusion

The prototype served to verify that the communication processors would function as expected, to provide rapid parallel processing of message traffic in the PARSOR architecture. Construction and testing of this prototype also led to refinements in the design. Thus, this prototype gives a clear demonstration that the concepts suggested in this architecture can be realized efficiently.

Previous chapters described the types of problems this architecture is designed to solve, the architecture itself, and the performance expected from it. The next chapter will address issues involved in using such a system, evaluate the

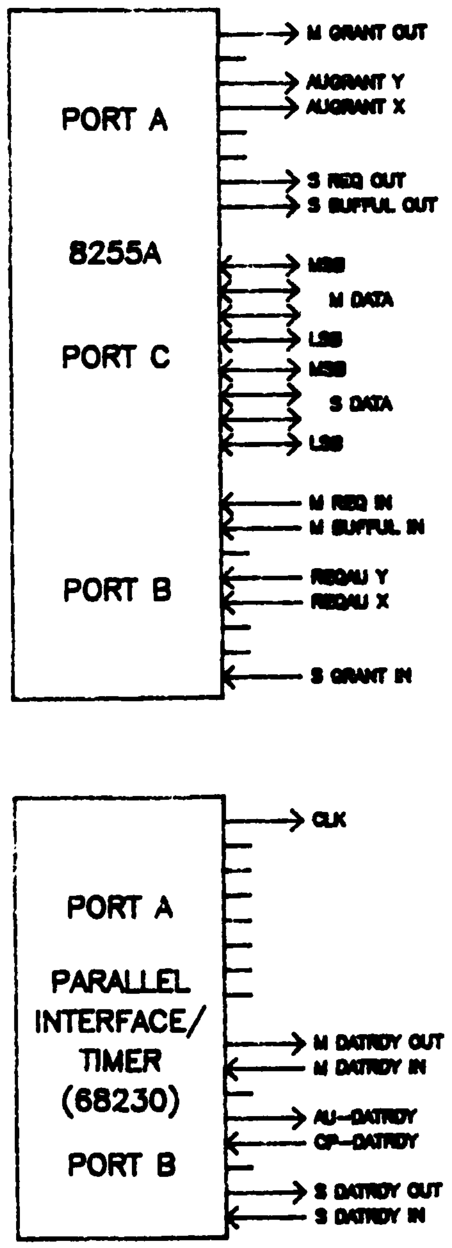


Figure 40 - ECB Interface Signals

overall system and its usefulness, and describe areas for further research.

CHAPTER SIX - DISCUSSION

Previous chapters have described the ParSOR architecture. They discussed the type of problem this architecture was designed to solve, presented the algorithm used in the system and the architecture of the system to execute this algorithm, and described the approaches used to verify the validity and capability of this architecture (i.e., by simulation and prototype design and testing). This chapter concludes the report with a discussion of the architecture, including issues of operational considerations, comparisons with other architectures, utility and potential capability of the design, and areas for further investigation.

Data Loading

Before the ParSOR system can begin the computations for a problem, data describing the problem must be loaded into the processors. The method of sending this data to the PEs has not been described in preceding chapters. This could be a bottleneck to the operation of the system. I have not designed a system in detail to perform this function, conducted simulations, or included such functions in the prototype. However, it is important to investigate this topic sufficiently to gain assurance that data can be loaded quickly and efficiently without major changes to the architecture which would affect its overall performance.

The data items to be loaded are as follows:

- Associative memory: pairs of indices of needed node values (j) and associated matrix entries ($a_{i,j}$).

- Message generator: number of messages to send, node number to use in messages, R and C for each message.

- Arithmetic Unit: number of terms required in each iteration, relaxation factor w, right hand side value b_i, parameter for convergence determination.

As with the rest of the architectural features of the ParSOR system, it is desirable in this case to maximize the amount of parallelism in the loading of data. Since each processor is to be implemented in a single chip, the addition of another data path to the chip for this data loading is very undesirable. The I/O pins used for this purpose would reduce the number of pins available for the use of the communication processors during computations, thus slowing those computations down. Thus, the data loading scheme should utilize the data paths provided for the exchange of data during computations.

Fortunately, the processors are already provided with extensive facilities for passing and processing messages quickly and efficiently. This suggests that data could be loaded via these same message passing facilities, if appropriate control structures could be devised. The message facilities for data loading should be implemented in a way which will not detract from the processing of messages for computations as described in previous chapters. One way to accomplish this is to add a single extra pin to each chip, and use it to signal whether the system is in the load mode or the run mode. This information could then be combined in the logic within each processing element, allowing the functioning to proceed as described in Chapter 3 while in the run mode, or as required to load data in the load mode.

Another possibility is to add a small number of mode control bits to each message. These bits could perform the same function as the mode bit, but would also allow other modes, to allow for functions such as convergence testing and routing final results of the computation back to the host. Combining these bits in the messages themselves may reduce the number of I/O pins, since the bits can be multiplexed over a smaller number of data lines. This approach also reduces the need for signals with very large fan-out loads, such as the mode pins of thousands of chips would present.

Figure 41 presents a sample message format which would allow for mode control as described above.

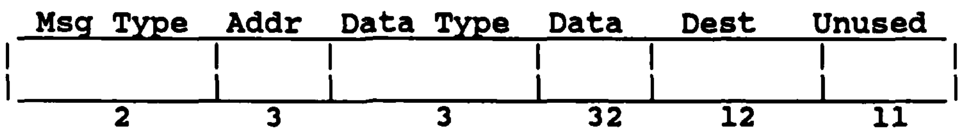


Figure 41 - Message Format for Data Loading

The numbers shown indicate the number of bits dedicated to each function. Note the total of 63 bits. This is a convenient number if the external bus must transmit data in three segments, and requires only one extra data line for each external bus. If the external bus were a different size, the message size could be modified to fit this case. This size is designed on the assumption that 60 bits are needed for the normal operating messages used in the system. The message with control bits must have more bits than this to allow a distinction between different message types while maintaining the ability to use any value within the data portion of the message.

Possible interpretation of the control bits is as follows:

- Message Type: 00 => Run
  01 => Load Data
  10 => Result for Host

- Address:	000 => AU
	001 => CP Up
	010 => CP Down
	011 => CP Left
	100 => CP Right
- Data Type:	000 => j, R, C
	001 => $a_{i,j}$
	010 => i, # messages for message generator to send.
	011 => b_i for AU
	etc.

Even though some data bits are marked unused, they could be used for certain functions. For example, in storing matrix entries in the conventional memory portion of the associative memory unit, it would be convenient to have an address provided, so the associative memory would not have to keep track of the number of items loaded to calculate this address. The extra bits could be used for this function. The presence of these unused bits also provides some flexibility in the arrangement of the data fields within the word, e.g., the Dest bits could be positioned where the values of R and C would go in the message for interprocessor communications, if this made the processing of the messages easier.

Using these control bits allows the required data to be loaded in the network of processors using the same communication scheme used for passing data messages during the computation. Clearly, however, the routing of the messages will be different. In normal operations, data is sent from the source PE to other PEs nearby. The routing information (R and C) is limited to 6 bits for each direction, which allows messages to be sent forward or backward only 32 rows or columns. For a large array, this is not far enough to load all the PEs with messages which originate at the edges of the grid.

This routing can be accomplished with the same basic resources available in the CP as described in Chapter 3 if the loading is accomplished using a separate interface processor for each column of processors, as shown in Fig. 42. Each of these units would be connected by a high speed data path to the host processor. The host would then send the data messages for loading the PEs over this path, with indicators for which column of processors they belong to. The loader would receive the messages intended for its column from this interface, and forward the data to the first PE in the column via the external bus interface. The load messages would always be sent over the external bus to CP Up. If the data needed to be sent to another node, it would be forwarded to CP down. Thus, the values of R and C need not be used to determine where to send the data, as

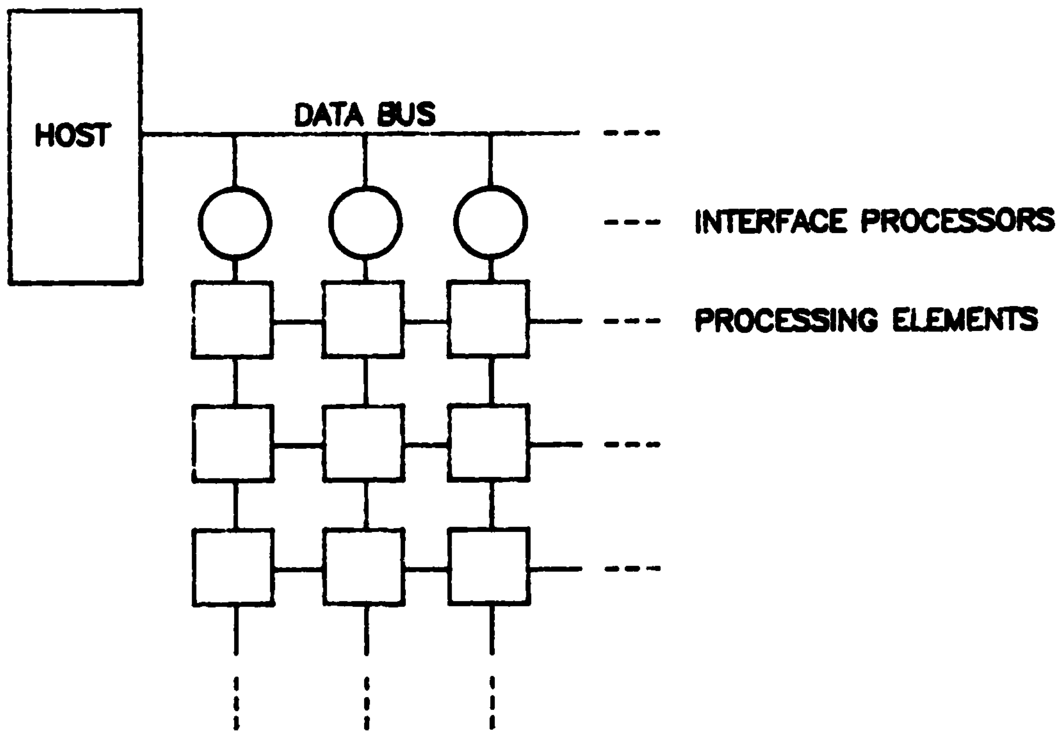


Figure 42 - Host Computer Interface

they are for normal operations. Therefore, the values of both R and C can be used to specify the number of rows of displacement to the destination PE. This would allow routing the data to as many as 4096 rows of processors, since 12 bits are available. The updating of R and C could still be performed by the RC update unit, with only minor modifications, since it has two adders available.

Once the data message arrives at the destination PE, the values of R and C will be 0. The Addr bits in the message can then be used to route the data to the particular CP (or the AU) for which it is intended. Within the CP, the data value would be presented at the inputs of all the storage locations which need such data. A controller would then examine the control bits and generate the necessary control signals to cause the data to be stored. A similar controller would handle storing the data in the AU, once it is sent over the AU interface.

Since the data is routed in only one direction during loading, the communication process can be speeded up somewhat from that used in normal operations. For example, only CP Up will send data over the internal bus, so the presence of the Load Data mode bits could be used to override the normal control of this bus and grant access immediately to that CP. Similarly, CP Up could gain immediate access to the AU bus. Thus, once a message arrives in INBUF for CP Up, the timing (expressed as a number of clock cycles) might be as follows:

0 - INFUL is set

2 - RC update unit completes updating of R and C (assuming they were not both zero).

3 - Data is moved to RTBUF if not destined for this CP (this happens after 1 clock cycle if R and C are zero in the incoming message). INFUL is cleared.

4 - Internal bus controller places data on bus and asserts IB-DATRDY.

5 - CP Down latches data from internal bus into IBBUF, and asserts ACK.

6 - CP Down moves data from IBBUF to OUTBUF (unless needed at that node).

7 - CP Down asserts DATRDY on external bus, sending first portion of message to next PE.

6 + C - INFUL is asserted at CP Up of receiving PE (C is number of cycles required to send a message over the multiplexed external data bus).

Once a message reaches the CP which the data is intended for, it should require only one more clock cycle to store the data in its destination. The control signals can be generated with combinational logic combined with the system clock. Thus, the storage operation requires little time.

The real key to performing this data loading quickly is the realization that the messages can be pipelined. Notice that once the data has been sent to RTBUF, INBUF is cleared. Thus, the INFUL flag stays high for only three clock cycles. As soon as this flag drops, the loader can begin sending the next message to that node. If it takes C cycles to send a message over the external bus, the INFUL flag will rise in the rising edge of the Cth cycle, and stay high for 3 cycles. Thus, the maximum rate for sending load messages is one every C + 3 clock cycles. For example, if each message is sent in three segments, a message can be sent to the first PE in the column every six clock cycles. The times required to send the message to another CP within the same node and store it there, or to store the data in CP Up, are less than or the same as the time to forward the message to another node, so this pipelining rate can be sustained during the entire data loading operation.

Some approximations are needed to determine how much data must be loaded. The data items to be sent to each node vary in length, so in some cases, several items can be combined in a single message. The total amount of data to be sent to each PE depends to some extent on the particular problem. For example, the number of RC pairs sent to the message generator depends on the number of messages that CP must send out. By the symmetry of the matrix of the problem to be solved, each CP will send data to as many nodes as it receives data from. Some messages serve to convey data to more than one node, since data routed to a distant node must pass through intermediate nodes to get there. Therefore, the number of RC pairs for messages sent out from a node is always less than or equal to the number of index values stored in the associative memory. Thus, the values of j, R and C can be combined in one message.

Each CP can require from zero to eight values of j, and corresponding values of $a_{i,j}$ (recall that each CP receives values of j for only one direction in the grid - thus up to 32 values of j can be stored in the entire Communications Unit). Assume that the average CP requires two such values. These two items will then require four messages for each CP (two containing tthe values of j and RC pairs for outgoing messages, and two more for the corresponding values of $a_{i,j}$). Another message for each CP can convey the node number of the variable computed at that node, and the number of messages to be generated by the message generator. In total, five messages are required for the CP data. The AU requires five messages to convey the values of the relaxation parameter, the value from the right hand side of the equation, the initial value of x for this node, the number of other node values required to complete an iteration, and the allowable change in the x value between iterations to determine when the iteration has converged. The total number of messages for each PE is then 25 (5/CP x 4 CPs + 5/AU). Each message is roughly 64 bits, or 8 bytes, in length.

A few more assumptions are necessary to determine the total time to load the network. Assume the network contains N^2 processors, arranged in an N by N grid. Also, assume that external busses are capable of sending a message in three segments. Thus, the loader will be able to send an entire message to the network every 6 clock cycles. The total load time will then be (6)(25)(N) or 150N clock cycles. The results of Chapter 4 indicate that each iteration requires approximately 300 clock cycles. Thus, the data loading takes about the same time as N/2 iterations.

This loading speed does not take into account the bandwidth required of the high speed interface to the host processor. Clearly, some additional information is required on this bus to route the messages to the correct column in the grid. For discussion purposes, assume a 50% data overhead. Thus, this bus must carry 12 bytes of data for each message. If a message is sent every six clock cycles to each column, and there are N columns, this bus must carry data at the rate of (12)(N)/(6) or 2N bytes per clock cycle (here, clock cycle refers to the clock cycle of the processor grid, not that of the data bus; the data bus may run on a different speed clock than that of the grid of processors). For large values of N, this is likely to prove to be the limiting factor in how fast data can be loaded. For example, if N is 100 (10,000 nodes in the processing grid), and the clock speed is 2 MHz, the interface would have to supply data at the rate of 400 MBytes/sec. Thus, even for this moderate sized problem and clock speed, the network is capable of absorbing data faster than most hosts are capable of supplying it.

There is nothing inherent in this data loading scheme which requires that all data be loaded for each run. For example, in impedance imaging, after the finite element problem is solved, the model is updated to account for the differences between computed and measured currents and voltages. However, this updating affects only the values of conductivity associated with each node, not the structure of the model, the connection patterns between nodes, or the boundary conditions (right hand side vector, b). Thus, only the matrix values would need to be loaded for successive runs - these amount to only 4% of the data which must be loaded for the entire problem, so the loading would be completed 25 times as fast.

Having the limiting factor in data loading be the host bus bandwidth is preferable to having the limitation in the processor grid itself. Several vendors (e.g., DATARAM of

Princeton, N.J.) supply wide bandwidth memories or data channels for minicomputers which can alleviate any bottlenecks which appear at this interface. If the limitation were in the ParSOR architecture, it might reduce the desirability of this architecture for some problems, since there would be nothing the user could do to overcome such a bottleneck.

The speed of data loading is compared with the overall computing speed later in this chapter.

Convergence Testing

As mentioned in Chapter 2, one of the problems with iterative algorithms is determining when the iteration has converged. This problem is especially difficult on parallel processors, since convergence tests must be applied to all the variables in the system. This requires long distance communications, which can slow down the performance of the processor substantially. The convergence test affects both the time required for a solution and the accuracy of the solution obtained. The time is a function of the number of iterations completed, which will be higher if the iteration test demands more accuracy. The accuracy depends on both requiring a certain degree of accuracy in the answer, and assuring that the test is applied correctly, so that it does not indicate convergence prematurely.

Since the problem to be solved is a linear system of equations of the form $Ax = b$, the ideal convergence test would be a measure of the error vector $b - Ax$. However, the computation of this vector is expensive in terms of time, so a simpler test is commonly used. This involves determining the vector norm $||x^{k+1} - x^{k}||$. Any vector norm may be used for this computation; a commonly used norm is the infinity norm, which gives the largest component of the change in x values, $\max (|x_i^{k+1} - x_i^{k}|)$. This norm is used because it is fast and easy to compute. The convergence test then involves comparing this vector norm with a prespecified limit, d. This parameter depends on the accuracy of solution desired and the nature of the data available. If the parameter chosen is too small, convergence might never be attained; if it is too large, the solution will not be accurate. To prevent excessive delays if the parameter chosen is incorrect or the problem fails to converge for some other reason, users often specify a limit on the number of iterations to be completed. For example, Kim [1982] specified a maximum of 500 iterations for his impedance imaging test runs.

One way to implement such a test on the ParSOR architecture would be to have each PE send its result to the host for each iteration, and wait for the host to signal whether to continue iterating or not. Such an implementation would be disastrous for the performance of the system - it would then spend the vast majority of its time transferring data and waiting for the host to determine whether the iteration had converged. Fortunately, there are many things which can be done to improve this situation.

After the AU finishes the computations for one iteration, it must wait some time until a neighbor node receives that result, completes its own iteration, and sends that value back to the first node to allow starting the next iteration. The AU can use some of this time to perform some of the tests needed to determine convergence. For example, it can calculate the difference between its most recent iterated value and the previous one, and compare this with a pre-stored limit. It can also update a counter to keep track of how many iterations it has completed, and store computed values.

It is not possible for each PE to determine when the entire problem has converged. The results of the computation at one node impact other nodes. Thus, node A may have a small change in x on one iteration, and think it has converged. But the value of x for node B could later change significantly, thus causing node A to change again. Thus, it is necessary to test all the nodes for convergence.

Saltz et al. [1986] propose an algorithm for convergence testing which minimizes the communication cost of this operation. They refer to this algorithm as asynchronous convergence checking, or ACC. At each node, the change in x is computed and compared with the limit d. Once the change in x is within the limit, the iteration number is noted. If the iteration stays converged for a certain number of iterations, a "convergence sequence" is said to have occurred; the iteration number of the first iteration in this sequence (noted earlier) is stored as the "header" of this sequence. These headers are routed to a central processor. When that processor has received a header value from every processor, it compares them, and selects the largest value as its "guess" of the iteration number at which the computation converged.

It is possible for a node which converged to diverge again, due to changing values from neighbors which have not yet converged. Therefore, the receipt of headers from all nodes is not sufficient to determine convergence of the entire problem. To check this, the central processor sends its guess of the iteration number for which the problem is converged to all the other processors in the system. If the iteration at a node which had converged diverges and then converges again, a new header will be generated. When the processors receive the guessed iteration number from the central processor, they respond by sending in their latest header value. The central processor then compares these with its last guess. If all the headers received are less than or equal to the guess value, all nodes were converged at that iteration. The central processor would then instruct all nodes to stop iterating and send in the value for that iteration. If one or more of the headers received at the CP is greater than the previous guess, the largest of the new values becomes the new guess, and the process is repeated until convergence is detected.

The ACC algorithm could be implemented in the ParSOR architecture either by providing a special purpose processor to do the global checks, or by using the host computer for this purpose. The need for this checking introduces extra message traffic on the external data busses between PEs. However, as shown in the previous chapters, this extra traffic has only a minor impact on the performance of the network. The delay to send the data to the host depends on the size of the grid of processors. Each processor in the path from a message source to its destination introduces approximately 10 clock cycles delay. Thus, if messages were sent back to the host by columns through the host interface, and there were 100 rows of processors, it would take approximately 1000 clock cycles to send the message from the most distant node to the host. This is roughly the time required to complete three iterations. The host interface units could keep track of the largest header received, and forward only this value to the host, thus reducing the number of values which the host must examine from one per variable to one per column of processors.

Once the host receives the indication that headers have been received from all the nodes, along with the values of the largest headers from each column, it could find the largest header, and quickly decide whether convergence had occurred. If not, it would again take about three iteration times to send the message to the most distant nodes. By the time this message got back to the most distant node, at least six, and more probably 10 or more additional iterations would have been completed. Thus, the nodes would be likely to have new header values ready, so the process could be repeated again soon.

This algorithm allows convergence testing to be completed in the ParSOR architecture in parallel with the computation. The impact of the global communication requirements on the speed of computations can be limited to the effects of the extra message traffic on the external and internal busses, which is moderate. The elapsed time, from the completion of the iteration on which convergence occurred until the time the host has the answer and knows that it is correct, will be related to the number of rows of processing elements times 10 clock cycles, plus the time the host requires to make the check of the header data. The total time required for convergence testing is likely to be on the order of the time required to complete a few iterations.

Another possible way to implement convergence testing would be to allow each PE to signal its convergence by a bit dedicated to this purpose. These bits would be logically OR'd together at a central point. When convergence was achieved, all the bits would be set, and all processors would be instructed to send in the results. This is the method used in the Finite Element Machine (see below). This would require one extra I/O pin per chip, and additional hardware to perform the OR operation for a large number of flags. It would reduce the message traffic required for convergence testing, and probably the time required for communication. I do not have data which would indicate a clear choice between these methods.

Overall Performance

Based on the discussion of the previous two sections and the simulation results shown in Chapter 4, it is now possible to get estimates of the overall level of performance achievable with the ParSOR architecture. As with previous such discussions, some assumptions are required for this purpose. The following estimates are based on the performance measured for the system for the test case which used 15 nodes, and long connection paths, as shown in Fig. 15 in Chapter 4. Assume that multiplication takes 20 clock cycles, and external bus messages take 3 cycles to send. This results in iterations which take 330 clock cycles each. Assume N processors are used, arranged in a square. Thus, from the section on data loading above, the data can be loaded into the array in $150N^{1/2}$ clock cycles (in the previous section, the number of rows was N, so there were N^2 processors). The total amount of data to be delivered is $2N^{1/2}$ bytes per clock cycle in order to load the array at this rate; if this rate can not be supported by the host, the loading will be slowed down accordingly.

The above parameters are functions of the size of the problem, which can be computed fairly simply. More difficult is the determination of factors such as how long the host will take to determine that convergence has occurred, and how many iterations are required to reach convergence. Researchers use a "model problem" consisting of solving the heat equation on a unit square, with a finite difference model and a distance of h between mesh points [Young, 1971]. The number of iterations to reach convergence for this problem is known to vary as 1/h, which relates to the square root of the total number of node values. For more complex problems, there is no such known result available, but heuristic results are less optimistic. Thus, one would expect the number of iterations to grow faster than the square root of the number of variables. This will also vary with the value of the relaxation parameter, and the accuracy of the initial values of the variables. Thus, it is impossible to predict in general the number of iterations which will be required for convergence.

The amount of time required for the host to examine the last set of headers it receives and determine that the problem has converged will vary greatly, depending on the capabilities of the host, the bandwidth of the data bus, other processes running on the host, and so on. For large problems, one would expect this to be smaller than the time required to complete a large number of iterations.

The estimated overall performance of the ParSOR architecture which follows is based on these assumptions:

- The model contains 10,000 nodes, in a 100 x 100 grid.

- 500 iterations are required to obtain convergence.

- Each iteration takes 330 clock cycles.

- The time to determine completion of the iteration is equal to the time required to execute 20 iterations. This is a difficult area to estimate, since no specific method has been proposed. This time could be more or less, depending on the method used.

- Case 1: The host has sufficient memory bandwidth to load data at the maximum rate the network can absorb.

- Case 2: The host can supply data at the rate of 20 MBytes/second.

- The ParSOR system runs at a clock rate of 2 MHz.

Using the above figures, the time required for the solution of the given linear system is:

Case 1 -	Loading	150(100) =	15,000	clocks
	Iteration	500(330) =	165,000	"
	Convergence Test	20(330) =	6,600	"

Total: 186,600 clock cycles, at 2 Mhz => 93.3 milliseconds.

Case 2 - Loading: $2N^{1/2}$ = 200 bytes/clock = 400 MB/sec = 20 times memory B/W, so loading takes 20 times as long, or 20(15,000) = 300,000 clock cycles.

Iteration and Convergence Test as above

Total: 471,600 clock cycles at 2 MHz => 235.8 milliseconds.

It is not possible to prove that this level of performance would be attained without actually constructing the array, since there are so many assumptions involved. However, the figures obtained above appear to be representative of the performance which could be expected with this architecture. In each case, fairly conservative estimates of performance, operations required, clock speed, and so on, have been used. While actual values of these parameters could vary significantly, there is no reason to expect that they should cause these results to vary by more than a small factor. Even if every aspect of these results is optimistic by a factor of ten, this system is still capable of solving a system of 10,000 variables in less than one second. Thus, this architecture holds the promise of very high performance.

For larger problems, the time to compute each iteration does not grow at all as the size of the grid grows, as long as the same interconnection pattern between nodes is used. Thus, the time required to solve larger problems would depend only on the growth in the time to load the data, and with the growth of the number of iterations required. The data loading time grows as $N^{1/2}$; as mentioned above, the number of iterations is likely to grow faster than this, so the rate of growth in the number of iterations required will determine the rate of growth of the solution time. But for the special case of the model problem, extending the problem above to a grid of 1000 x 1000 processors would increase the time by only a factor of 10, giving a solution time of roughly 2.5 seconds for a linear system of 1,000,000 variables.

Summary Of ParSOR Architecture

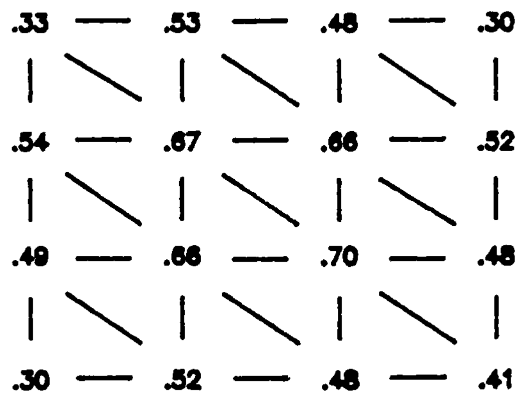
The ParSOR architecture can be categorized as an MIMD architecture, since each processor operates independently, and there is no central controller. However, the functions of each processor are built into the hardware, so there is no instruction fetching, and thus no "instruction stream". Rather, the appropriate operations commence as soon as the required data items arrive. In this sense, this architecture is analogous to that of a data flow machine. However, in a true data flow architecture, the operations which are ready to commence because all data is available can be assigned to any processor, so a processor will never sit idle while several data items await processing at another processor, as sometimes happens in ParSOR. Thus, as with most parallel processors, the ParSOR architecture has unique features which cause it not to fit exactly into any particular category.

The architecture achieves high degrees of parallelism in many ways. The multiple Arithmetic Units allow many arithmetic operations to take place in parallel. The partitioning of the processor into an Arithmetic Unit and a Communications Unit allow computation and communication to take place in parallel. The four separate Communication Processors allow parallel communications with four neighbors from each node. The independent functional units within the CP allow parallel handling of the different actions required to process incoming messages. Finally, the proposed convergence testing scheme allows convergence testing and iterations to take place in parallel.

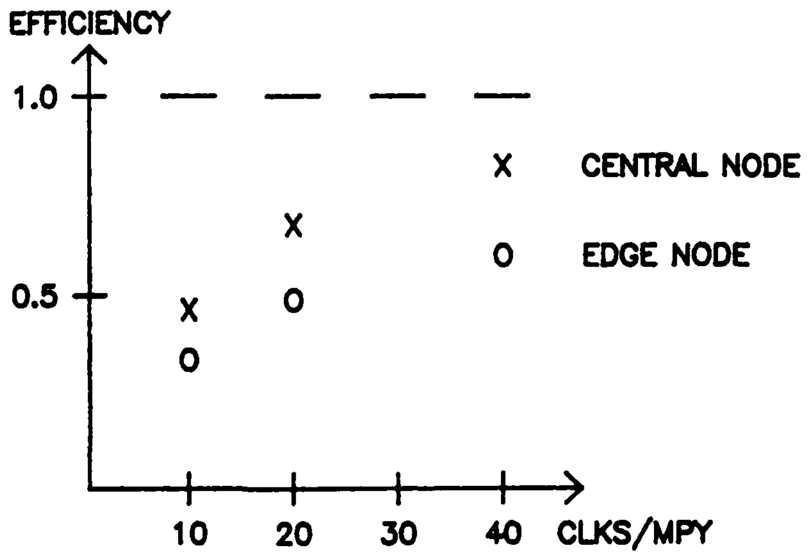
One concern with this amount of parallelism is the possibility that too many processors are available. Using a multicolor node ordering with C node colors, for example, it is clear that each node can be updated only every C steps. Then it would seem that if a processor were dedicated to the computations for only one node, it would necessarily be idle for C - 1 out of every C steps. Thus, for a three color ordering, each processor would be idle two thirds of the time. However, this does not turn out to be true. An examination of the computation used in the SOR algorithm shows that the computation can be performed in increments as data items become available, and part of the computation can be done based only on data available as soon as the processor finishes its previous computation. Thus, if the communication network can keep the data supplied, there is much computation which can be completed before the last data item becomes available. As long as some of the processors in the grid are kept busy at least 50% of the time, the claim that the same speed can be attained with a smaller number of processors is clearly false.

To test whether this was the case in this architecture, the simulation program was modified to keep track of the portion of the time that the multiplier was busy during a run. For this test, the problem illustrated in Fig. 16a, namely 16 nodes in a four by four grid, with triangular elements interconnecting them, was used. This model was run for three iterations, and the portion of the time each multiplier was busy (assuming 20 clock cycles per multiplication) was calculated. Even the first iteration, which performs fewer multiplications than later iterations, was included in the calculation. The results are shown in Fig. 43a, with the number at each point giving the fraction of the time the multiplier at that point in the grid is busy. Several processors are busy with computations over two thirds of the time, rather than the maximum one third as described above. Thus, given the same multiplier speed, it is clearly not possible to perform this computation with equal speed with a smaller number of processors. Figure 43b shows the variation of these results as the speed of the multiplier varies, for both a central node in the grid, and a node on the edge, which has fewer multiplications to perform. As expected, the efficiency is lower for a faster multiplier, since it takes less time to perform its computations, and thus more time waiting for data. Even with a fast multiplier, the central nodes are busy a substantial portion of the time, which indicates that the communication scheme provides data to the processors effectively.

The architecture is modular, and thus easily expandable. The processors along the edges of the array are identical to those in the center. The operation of each CP



A



B

Figure 43 - Processor Utilization

is determined by the values which are loaded into it at the beginning of the run. Thus, the CPs on the edges of the array which do not have any neighbor to connect to can simply be programmed to send out no messages. The interconnections between PEs on a board are very regular, and should be easy to design into printed circuit boards. The connections between boards are equally simple and regular, but do involve a large number of signal lines (assuming there are many PEs on each board). Adding more processors to a system is a simple matter of plugging in the additional boards - no hardware or software modifications to the existing boards are required.

The ParSOR architecture is closely matched to the requirements of the SOR algorithm, for problems in which the problem nodes are located close to the nodes to which they are connected, and for which the system matrix is symmetric. This causes it to be impossible to use the same architecture for more general purpose computations. This was a deliberate tradeoff - my goal was to design the fastest possible architecture for a specific type of problem. Making this architecture more general purpose would have reduced the amount of parallelism in its operation substantially, and thus reduced its performance capability on the type of problem in which I was interested.

The usefulness of such an architecture would depend on the needs of the user. The architecture will not work if there are not enough processing elements to handle all the variables. Thus, a user who needs to solve problems with 15,000 variables can not choose to buy only 1,000 processors and wait longer for results. Furthermore, if that user does buy 15,000 processors, and later encounters a problem with 16,000 variables, he again needs to buy more processors. Thus, this architecture requires the user to be sure to have enough processors to handle his problems.

While these considerations make the ParSOR architecture undesirable for some applications, they result directly from the attempt to gain the fastest possible solution for the type of problems described in Chapter 1. This system is certainly not appropriate for all applications involving the solution of large linear systems. However, the design is consistent with the goals for the system, as stated in Chapter 1.

Comparison With Other Architectures

The ParSOR architecture uses parallelism to obtain a fast solution of large, sparse SPD systems of linear equations. Several other architectures have been proposed for this purpose, or are applicable to this type of problem. Among these are the Finite Element Machine, the Sparse Matrix Solver Machine , and an implementation of a technique called Parallel Nested Dissection on a parallel processing array called the Thinking Machine. In this section, the ParSOR architecture is compared to these machines, and also to the architecture of a single chip computer designed specifically to be interconnected into large parallel ensembles, known as the Transputer.

The Finite Element Machine is designed specifically to perform finite element method computations [Jordan, 1978a, 1978b]. This architecture provides for 1024 processors arranged in a grid, with each one connected directly to its eight nearest neighbors, and also to a global data bus. There is a single central controller for input and output, but otherwise, the processors run independently. Each processor consists of a commercially available microprocessor, RAM and ROM, a floating point coprocessor, multiple I/O ports, and signaling circuitry.

Each processor is connected directly to its eight nearest neighbors in a two-dimensional grid for data communication purposes. A global data bus, connected to all processors, is provided to allow processors to communicate with others which are not among their 8 nearest neighbors. Each processor can be programmed to perform the computations for any subset of the nodes in the problem. Each processing element includes a general purpose microprocessor, with floating point hardware, so there is a great deal of flexibility in the use of this network.

To expedite the convergence checking process, each processor has a signal line to a central host. When the processor determines that all the variables which it computes have converged, it sets this flag. All these lines are OR'd together at the central host, so the host can detect apparent convergence with almost no communication delay. Time is still required to retrieve the results.

This system is intended to implement all aspects of finite element method computation on a distributed processor, so it includes more functions than the ParSOR architecture. The architecture should give excellent performance for this purpose, if the problems of supplying and interconnecting this much hardware can be solved. However, it has some drawbacks. Since the processors are commercial microprocessors, a certain amount of software overhead for instruction fetching is inevitable. The parallelism between computation and communication used in the ParSOR architecture can not be achieved. In fact, if more processors are added (beyond a certain number) to solve a finite element problem involving a given number of nodes, the computation time will actually increase due to the communication overhead [Adams and Crockett, 1984].

Also, the architecture does not allow easy addition of more nodes to increase processing power. The design approach used is to provide a large array of powerful processors, which provides good speed for large problems.

As problem size grows, the same network can be used, but the solution will take longer. On this sense, this system is similar to a conventional uniprocessor. However, if 1024 processors do not provide sufficient speed, the architecture can not be easily extended to increase this speed. Thus, the price of solving larger problems is slower solutions. Many users would probably prefer this to the price of having to buy more processors for larger problems. However, for those who need the best possible speed for very large problems, the ParSOR system provides a clear alternative.

In the Finite Element Machine, the cost of each processor is high, since each one is complex, including many ICs for the main processor, floating point coprocessor, memory, communications channels, and supporting functions. Since the ParSOR processor is implemented on a single IC, it should be possible (assuming more than a few such machines are produced) to build several processors for the cost (including power, board space, interconnections, and so on) of a single processing element in the Finite Element Machine. Thus, for the same cost, the ParSOR system could be built to handle a reasonably large number of nodes. At a higher cost, it could be expanded to handle very large problems with little increase in processing time. For some applications, this could be a desirable property.

Amano, Yoshida, and Aiso [1983] proposed an architecture which they called the Sparse Matrix Solving

Machine, or $(SM)^2$. This machine is designed to solve linear systems by an iterative technique, in order to preserve the sparse nature of the system. The design is based on some general observations about the computations involved, particularly the data communications required. In particular, the authors point out that most data communications are localized. However, they exploit this locality of data sharing in a very different way than the ParSOR architecture does.

In this system, communication is accomplished by shared access to central memory. For a large array of processing elements, this scheme leads to severe conflicts, as a large number of processors attempt to access memory at the same time. The authors claim that this problem can be eliminated by exploiting the locality of data communications inherent in the problem. They propose a memory system which keeps multiple copies of data items, allowing several processors to read the same data item at one time. By grouping the processors in "clusters", and allocating more memory copies for data items within a cluster than for items from distant clusters, they claim to give processors rapid access to needed data without having to maintain an excessive number of copies of all data items.

This design directly addresses the same problem I am interested in. However, some aspects of the design are of concern. With N PEs in each cluster, N copies of each data item for that cluster are provided. In addition, M $(M < N)$ copies of the data for the cluster to the left, and M copies of the data for the cluster to the right are provided, along with one copy of the data for each of the K other clusters in the system, for a total of $N + 2M + K - 3$ copies. The authors suggest keeping M equal to 1 to minimize the complexity of the controller. Thus, even with the data clustering scheme, a system with 1000 processors distributed over 20 clusters must retain 69 copies of each data item. But for a typical finite element problem, only 5 to 10 copies of the item will actually be used by other processors (Chapter 2). Thus, this design requires the transmission and storage of much redundant data.

A second concern is with the expandability of the system. For each new cluster of processors added, a separate external data bus must be added to the system to transfer data from that cluster to the existing clusters. If the memories of the existing clusters are not large enough to handle this data, they must all be expanded. The authors show that about 50 processors per cluster is the upper bound for effective performance. Thus, to allocate 10,000 processors to a problem would require 200 external busses, and 249 copies of each data item. Thus, expansion of the design to this size, or even larger sizes, would be extremely expensive, and result in very high redundancy of storage and communication facilities.

The authors do not give any demonstrated or simulated performance results for their system. The performance estimates they provide are based on an assumed computational speed of the processors, and are given in terms of a number of operations executed per second. The authors do not discuss the time required for their machine to solve the type of problem for which it is designed. Thus, it is not clear how much this architecture contributes to the rapid solution of sparse linear systems.

Pan and Reif [1985] implemented an algorithm based on parallel nested dissection on the Connection Machine [Hillis, 1985], built by Thinking Machines, Inc. Since the algorithm used is a direct solution algorithm, this system avoids the problems of choice of a relaxation parameter and convergence testing inherent in the SOR method. However, it does suffer from the increased storage requirements and numerical stability problems inherent in the application of a direct solution technique.

The Parallel Nested Dissection algorithm numbers the nodes in the problem in such a way that the system matrix can be partitioned into several submatrices which are not connected to each other. Thus, the submatrices may be reduced independently, and in parallel. Once this is done, the results of these reductions must be combined to obtain the overall solution. In this implementation, this is accomplished by forwarding the intermediate results to additional processors to generate the overall result. This requires more processing elements than there are variables in the system.

The authors claim that for a planar model with N variables, $N^{1.5}$ processors are required. The machine on which this algorithm is being implemented contains 64,000 processors, but with this algorithm, this machine could solve problems of no more than 1,600 variables. Thus, expandability of this architecture to very large problems would require an excessive number of processing elements. For example, to solve problems involving 10,000 nodes, a total of 1,000,000 PEs would be required.

Recognizing this problem, the authors devised a variation of their algorithm, which will run in N processors for N variables, but will require additional time to compute the solution. The time required is still small, growing as $N^{1/2}$ for the optimum case of a square grid of nodes with only nearest neighbor connections. For other types of problems, the grid of nodes can not be subdivided as far, so less parallelism is present, and the solution is slower. However, the algorithm is being implemented in a general way, which will solve the less than optimal cases, at the expense of more processing time.

The algorithm is a direct solution method. Mathematically, the solution to the problem $Ax = b$ can be stated immediately as $x = A^{-1}b$. The computation of the inverse of A is generally considered very impractical, since this matrix will in general be dense, and thus require a huge amount of storage for a large system (e.g., a system with 10,000 variables would require 10^{10} storage locations). The parallel nested dissection algorithm instead computes a series of factors which can be multiplied together to obtain A^{-1}, while still maintaining much of the sparsity of the original matrix. Thus, the algorithm works in two stages: first, the factorization of A^{-1} is computed, then b is multiplied by this series of matrices to obtain the solution vector, x.

This approach has the advantage, mentioned in Chapter 2, that if the same system must be solved several times, with only the right hand side (vector b) changing, the factorization can be precomputed, and the successive right hand sides can be solved quickly. For example, in stress analysis problems, the designer might wish to determine the effects of different or time varying loading on a structure. The loading parameters would be contained in vector b, and the structure itself, represented by matrix A, would not change. Thus, the factorization for A could be used to solve for the effects of loading very quickly.

However, in problems for which any element of the A matrix changed between runs, the entire factorization would have to be recomputed. This is the case in impedance imaging - after the current distributions are found by solving the linear system, the impedances of the elements are updated (scaled) before the next run. Thus, a complete factorization and solution would be required for each step, if a direct solution method was used. Parallel nested dissection is less effective for this type of application. The ParSOR architecture, on the other hand, can effectively exploit the information available in this application. Since the structure of the matrix would not change between runs, and small changes in the element parameters are likely to cause only small changes in the solution, successive runs could be performed with only a small amount of data loading, and the results from the previous run could be used as a starting point, to reduce the number of iterations required.

The Connection Machine, on which this algorithm is being implemented, consists of 64,000 processors, interconnected by a hypercube network. The algorithm requires both nearest neighbor (grid) communication, and tree type interconnections, which can be implemented using the hypercube. Each processor in this network is a bit serial processor, and is provided with a significant amount of storage (4 kbits on the standard model, with more storage optional). The use of bit serial processors keeps each processor simple, but makes the operation slower. From simulation results, the authors estimate that the solution of a system of 16,000 variables would require 20 to 30 seconds for the factorization, and 2 to 3 seconds for the solution. For problems which could make repeated use of the factorization, this would give very fast response, e.g., a designer performing stress analysis on a structure could change the loading and see the results of this change within seconds.

This algorithm could be implemented on a more suitable architecture, and give even better performance. Each processor could be supplied with a reasonably fast multiplier, which would result in much faster computation. There are two difficulties to be overcome in this design: each chip would require a fairly large memory to store the filled-in values in the matrix, and the interconnections between CPs would need to support both near neighbor grid and tree structures. The storage problem is aggravated by the numerical stability problem of the direct solution method (described in Chapter 2) - this requires longer word lengths to achieve acceptable accuracy. Thus, the processor used in this architecture is likely to be more complex, and therefore more expensive, than the ParSOR processor. For some applications, this could be the most desirable system for solving the types of problems encountered.

Another possible application of commercially available hardware is to construct a parallel processing grid using the INMOS Transputer [Walker, 1985] for the processing element. The Transputer implements a small microprocessor, memory, and four I/O ports on a single chip. In this sense, it is very similar to the ParSOR architecture. However, the Transputer is intended to be a general purpose, programmable processor. The ALU is stack oriented, rather than register oriented. Thus, operands are taken from the top of the stack, rather than registers or memory. This allows the opcodes to be short - only the operations of loading data on the stack and storing data from the stack need to specify addresses.

Each processor contains communication paths to its four nearest neighbors. Separate data lines are used for bit serial transmission of data into and out of the port for each neighbor. Thus, the processor can simultaneously send and receive data in each of four directions. Each port uses a programmable DMA channel to store data received in (or read data to be sent from) memory. The processor needs only to program these channels to handle the data communication, then it can go on to other processing tasks while the channels complete the communication. The chip generates a special extra high speed clock (up to 80 Mhz) to transmit messages quickly over these links, even though the data is sent one bit at a time. The programming language for this system, Occam [INMOS Limited, 1984] includes specific constructs to allow processes to wait for and send data. This simplifies programming parallel processes which depend on data from each other.

The advantage of this system for solving large sparse linear systems is that the chip is already commercially available. This saves the cost of designing the actual chip to be used in the system. Commercial sales of the chip are likely to keep the cost of each chip low. As with other general purpose systems, a network of such systems would have utility beyond the solution of sparse linear systems.

The disadvantages of this system come from its general purpose nature. Much of the parallelism designed into the ParSOR architecture is not available with this system. For example, the links can continue receiving messages while the processor is busy doing multiplication. However, they are not capable of determining what to do with the data they receive. For each message received, the processor must stop performing computations, examine the message, determine what needs to be done with it, and if necessary, program another channel to send it on. Thus, the processor will be kept busy a significant amount of the time handling data communications. Since the processor is software driven, it must perform software overhead functions, such as saving the status of the computation in progress in order to examine a data item received, and then restoring that status information on return from the data examination procedure. This will cause a further reduction in performance as compared to the ParSOR architecture. Also, the use of a general purpose chip eliminates the possibility of tailoring the subsystems on the chip to the requirements of the problem. For example, the Transputer chip uses a simple and small processor, which implies slow multiplication. Thus, it is not possible to redesign the chip with a higher performance multiplier and still retain the advantages of using a commercially available chip.

Thus, while the Transputer is capable of fast solution of large, sparse linear systems, it is not the optimum architecture for this purpose.

Future Research

As with most research activities, this research has raised new questions as well as answered old ones. Described below are some areas for further investigation.

Before an actual chip can be designed, a particular multiplier must be selected. Several types of multipliers have been designed for various applications. These involve a tradeoff between hardware complexity and solution speed. Additional hardware raises the cost of the system by increasing the design cost, increasing the area required for layout, increasing the power requirements, and reducing the yield of the manufacturing process. The results of Chapter 4 give an indication of the effects of a slower or faster multiplier on the performance of the system. Specific multiplier designs need to be studied, so that one can be selected for this application with all these factors in mind. This selection is likely to take place well into the design process, since it will depend to a great extent on how much logic can be accomodated in the technology chosen for implementation, and how much of this capacity is consumed in the design of the CPs.

Detailed design is also needed for the rest of the AU. In order to know when it has finished an iteration, it must count the number of data items it has already incorporated into its computation. It needs a way to select from among several possible CPs which desire to send data at the same time, and needs a means of storing data items received, and moving them to the multiplier when it is available. It also needs some logic to implement the convergence testing scheme to be used. None of these functions is complicated to implement, but the details would need to be worked out before the actual construction began. Also, it would be important to determine whether the wordlength of 32 bits is appropriate. This length was chosen based on the numerical stability of iterative solution methods, and used the assumption that the problem would be scaled so that the numbers in the matrix would not have magnitudes greater than one. Thus, I assumed a 32 bit fixed point word. Some study is required to determine whether this would provide adequate accuracy and stability for real applications.

In order to design the AU in detail, it is necessary to verify the efficacy of the convergence testing scheme described above. This would require studying the algorithm in more detail, and perhaps performing simulation studies to verify its cost. This would require a simulation program constructed very differently from the one I wrote, since the execution time of my program would be prohibitive for this application. The ACC algorithm has been tested on a somewhat different parallel processor with good results, so it is the implementation of this algorithm on the ParSOR architecture which needs to be studied, rather than the algorithm itself. In particular, it is necessary to study the impact of the extra message traffic on the performance of the normal computations of the network. There is extra capacity in the communications scheme, so the message traffic for convergence checking can take place without totally blocking other types of communication, but there will be some extra delay due to this extra traffic.

This paper has not covered any of the aspects of the design of the interface processors, which connect the ParSOR system to the host computer. Some basic functions of these processors have been described, but the exact functions required have not been established. For example, their contribution to the convergence checking algorithm needs to be determined. Their interface to the ParSOR system can follow closely the design of the Communications Processor, but their interface to the host data path would depend on the characteristics of the particular host used. This device would have to be designed in order to implement the overall system.

Once the design of the chips is completed, the clock speed of the chip could be determined. This has a direct effect on the performance of the system for the solution and convergence testing portions of the operation. As described above, the data loading in many cases is likely to be limited by the bandwidth of the host interface. For a processor array of N columns, this interface must supply approximately 2N bytes of data per clock cycle. As the clock speed increased, this required data speed would increase with it. For applications whose data rate is limited by the bandwidth of the interface, a faster clock will not speed up the data loading. However, it will speed up the computation linearly, since all performance parameters are computed in terms of the number of clock cycles. For applications with reduced data loading requirements (such as when the same data structure can be used for successive runs), a faster clock would provide significantly faster execution. It would also be of benefit for large problems or those whose nature require a large number of iterations, which would cause the solution time to be substantially longer than the data loading time.

Some hardware considerations must be evaluated if this system is to be implemented. These include factors such as the number of processors which could be accomodated on a single circuit board, the interconnections between the boards, supplying a clock signal to a large number of processors, and so on. The individual processors are designed so that additional rows and columns of PEs can be added with no modifications to the existing ones. Ideally, the circuit boards containing the chips should be designed in such a way that additional boards can be simply plugged into the array, thus expanding it to handle larger problems.

One possible extension to this architecture would be to allow it to work with vector quantities. The finite element method can be used to solve problems involving vector fields, such as stress and strain, and complex quantities, such as voltage and current phasors used in power system modeling, in addition to scalar fields such as temperature and electric potential. Extending the ParSOR architecture to this case would not require any fundamental changes. The messages would have to be longer to accomodate the two or more dimensions of each vector quantity, and the AU would have to be modified to perform vector operations such as addition and multiplication. In other respects, the operations to be performed at each node, particularly the communication processing, would be the same. This change would make the system useful for a wider range of tasks.

Another extension would be to modify the architecture to implement three-dimensional grids of processors. The largest finite element problems and other linear systems tend to result from three dimensional models. For example, the atmospheric model for weather forecasting is a three dimensional model. Kim, Fahy and Tupper [1986] suggest a three dimensional body model for impedance imaging with over 100,000 elements. It is impossible to map a large three dimensional grid of nodes into a two dimensional grid of processors and still keep the connected processors close together. In the ParSOR architecture, the performance suffers if connected nodes are distributed over long distances in the processing array.

The architecture of the processing element in this system was specifically chosen to be easily expandable into three dimensions. This would simply require the addition of two more Communication Processors in the CU (to handle large numbers of nodes, the number of bits allocated to j would have to increase, but this is a design detail, rather than an architectural one). These processors would communicate with the layers of processors above and below the layer containing that processor. The data routing scheme would have to be modified to include this extra dimension, but this could be done quite easily. In three dimensional models, each node has more near neighbors to connect to, so there would be more computation required for each iteration.

This might cause the designer to reduce the number of pins in the external bus, in order to keep a good match between communication and computation speeds, and to simplify the interconnections between layers of processors. More simulation would be required to determine the performance of such a system. However, it could be expected to be just as viable a candidate for the parallel solution of these problems as the current design is for two dimensional problems.

Conclusion

This thesis has presented an architecture for the rapid solution of large, sparse, SPD systems of linear equations. This architecture allocates a separate processor for each variable in the system to be solved. The processors are connected in a two dimensional grid, with direct data paths from each processor to its four nearest neighbors. Attached to each communication path is a special purpose communication processor, capable of analyzing the data received and routing it to its destination without intervention from the arithmetic processor. This architecture allows a high degree of parallelism within the operation of each PE in the system, in order to execute the SOR algorithm as quickly and efficiently as possible. Extensive and detailed simulations of the proposed architecture have been performed, and verify its fast and efficient operation. A prototype of the processing element has been built and tested, in order to verify that the functions could be realized in actual hardware.

The results of these tests show that this system can provide very high levels of performance for the intended applications, comparable with or superior to the fastest methods of solving such systems yet proposed.

BIBLIOGRAPHY

Adams, L.M. and T.W. Crockett, "Modeling Algorithm Execution Time on Processor Arrays", Computer, Vol. 17, No. 7, July 1984.

Amano, H., T. Yoshida and H. Aiso, "$(SM)^2$:Sparse Matrix Solving Machine", The 10th Annual International Symposium on Computer Architecture Conference Proceedings, pages 213 - 220, IEEE Computer Society Press, 1983.

Axelsson, O. and V.A. Barker, Finite Element Solution of Boundary Value Problems - Theory and Computation, Orlando, Florida:Academic Press, 1984.

Birtwistle, G.M., O-J. Dahl, B. Myhrhaug and K. Nygaard, SIMULA Begin, New York: Petrocelli/Charter, 1973.

Carre, B.A., "The Determination of the Optimum Accelerating Factor for Successive Over-Relaxation", Computer Journal, Vol. 4, 1961.

Cuthill, E. and J. McKee, "Reducing the Bandwidth of Sparse Symmetric Matrices", Proceedings of the 24th National Conference of the ACM, New Jersey: Brandon Systems Press, 1969.

Duff, M.J.B., "Review of the CLIP Image Processing System", AFIPS Conference Proceedings - 1978 National Computer Conference, Montvale, New Jersey: AFIPS Press,1978.

Duff, I.S., R.G. Gains, J.G. Lewis and H.D.Simon, "The Harwell-Boeing Sparse Matrix Collection", to appear in SIAM Journal on Scientific and Statistical Computing.

Feng, T-Y., "A Survey of Interconnection Networks", Computer, December 1981.

Franta, The Process View of Simulation, Amsterdam: North-Holland, 1977.

Gannon, D., "A Note on Pipelining a Mesh Connected Multiprocessor for Finite Element Problems by Nested Dissection", 1980 Conference on Parallel Processing Conference Proceedings, IEEE Press, 1980.

George, A. and J.W. Liu, Computer Solution of Large Sparse Positive Definite Systems, Englewood Cliffs, New Jersey: Prentice-Hall, 1981.

Golub, G.H. and C.F. Van Loan, Matrix Computations, Baltimore: The Johns Hopkins University Press, 1983.

Gottleib, A., R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph and M. Snir, "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer", IEEE Transactions on Computers, Vol. C-32, No. 2, February 1983.

Hagemen, L.A. and D.M. Young, Applied Iterative Methods, New York: Academic Press, 1981.

Hennessy, J.L., "VLSI Processor Architecture", IEEE Transactions on Computers, Vol. C-33, No. 12, December 1984.

Hillis, W.D., The Connection Machine, Cambridge, Massachusetts: The MIT Press, 1985.

Huebner, K.H. and E.A. Thornton, The Finite Element Method for Engineers, New York: John Wiley & Sons, 1982.

Hwang, K. and F.A. Briggs, Computer Architecture and Parallel Processing, New York: McGraw-Hill, 1984.

INMOS Limited, IMS T424 Transputer Reference Manual, 1984.

INMOS Limited, Occam Programming Manual, Englewood Cliffs, New Jersey: Prentice-Hall, 1984.

Jennings, A., Matrix Computation for Engineers and Scientists, London: John Wiley & Sons, 1977.

Jordan, H.F., "A Special Purpose Architecture for Finite Element Analysis", Proceedings of the 1978 International Conference on Parallel Processing, IEEE Press, 1978.

Jordan, H.F., "A Special Purpose Architecture for Finite Element Analysis", ICASE Report 78-9, Institute for Computer Applications in Science and Engineering, Hampton, Virginia, 1978.

Kim, Y., W.J. Tompkins and J.G. Webster, "A Three-Dimensional Modifiable Body Model for Biomedical Applications", IEEE Frontiers of Engineering in Health Care 3, 1981.

Kim, Y. A Three-Dimensional Modifiable Computer Body Model and Its Applications, PhD Dissertation, University of Wisconsin - Madison, 1982.

Kim, Y., J.G. Webster and W.J. Tompkins, "Simulated and

Experimental Studies of Temperature Elevation Around Electrosurgical Dispersive Electrodes", IEEE Transactions on Biomedical Engineering, Vol. BME-31, No. 11, November 1984.

Kim, Y., J.B. Fahy and B.J. Tupper, "Optimal Electrode Designs for Electrosurgery, Defibrillation and External Cardiac Pacing", submitted to IEEE Transactions on Biomedical Engineering, March 1986.

Kung, H.T. and C.E. Lieserson, "Algorithms for VLSI Processor Arrays", in Introduction to VLSI Systems, Reading, Massachusets: Addison Wesley, 1980.

Kung, H.T., "Why Systolic Architectures", Computer, January, 1982.

Kung, S.Y., H.J. Whitehouse and T.J. Kailath (editors), VLSI and Modern Signal Processing, Englewood Cliffs, New Jersey: Prentice-Hall, 1985.

Mariani, M.P. and E.J. Henry, "PEPE - A User's Viewpoint; a Powerful Real Time Adjunct", AFIPS Conference Proceedings - 1978 National Computer Conference, Montvale, New Jersey: AFIPS Press,1978.

Pan, V. and J. Reif, "Efficient Parallel Solution of Linear Systems", Proceedings of the 17th Annual ACM Symposium on Theory of Computing, Providence, Rhode Island, 1985.

Potter, J.L., "Image Processing on the Massively Parallel Processor", Computer, January 1983.

Saltz, J.H., V.K. Naik and D.M. Nicol, "Reduction of the Effects of the Communication Delays in Scientific Algorithms on Message Passing MIMD Architectures", ICASE Report No. 86 - 4, Institute for Computer Applications in Science and Engineering, Hampton, Virginia, 1986.

Schendel, U., Introduction to Numerical Methods for Parallel Computers, Chichester, England: Ellis Horwood Limited, 1984.

Siegel, H.J., "Interconnection Networks for SIMD Machines", Computer, June 1979.

Siegel, H.J., "PASM: A Partitionable Multimicrocomputer SIMD/MIMD System for Image Processing and Pattern Recognition", Technical Report - EE 79-40, School of Electrical Engineering, Purdue University, Lafayette, Indiana, 1979.

Snyder, L., "Introduction to the Configurable, Highly

Parallel Computer", Computer, January 1982.

Stewart, G.W., Introduction to Matrix Computations, New York: Academic Press, 1973.

Tanimoto, S.L., "A Pyramidal Approach to Parallel Processing", Proceedings of the 10th Annual Symposium on Computer Architecture, Sweden, 1983.

Uhr, L., Algorithm-Structured Computer Arrays and Networks - Architectures and Processes for Images, Percepts, Models, Information, Orlando, Florida: Academic Press, 1984.

Vick, C.R. and J.A. Cornell, "Pepe Architecture - Present and Future", AFIPS Conference Proceedings - 1978 National Computer Conference, Montvale, New Jersey: AFIPS Press,1978.

Walker, P., "The Transputer: A Building Block for Parallel Processing", Byte, Vol. 10, No. 5, May 1985.

Weedy, B.M., Electric Power Systems, New York: John Wiley & Sons, 1979.

Westlake, J.R., A Handbook of Numerical Matrix Inversion and Solution of Linear Equations, New York: John Wiley & Sons, 1968.

Wilkinson, J.H., Rounding Errors in Algebraic Processes, Englewood Cliffs, New Jersey: Prentice-Hall, 1973.

Woo, H.W. and Y. Kim, "A Review of Electrical Impedance Imaging Techniques in Medical Body Imaging", submitted to IEEE Transactions on Biomedical Engineering, 1986.

Young, D.M., Iterative Solution of Large Linear Systems, New York: Academic Press, 1971.

APPENDIX A - Listing of BASIC Demonstration Program 1

```
1 '  Program  to  demonstrate  operation  of  parallel  SOR
2 '  algorithm  with  1  PE per node.  This example  has  36
     nodes, connected like
3 ' FEM triangles:  1 - 2 - 6
4 '                 | \ | \ |
5 '                 3 - 5 - 7
6 '                 | \ | \ |
7 '                 4 - 8 - 9   etc.
8 '  Notice that the numbering scheme is a front that  moves
9 '  in the direction of  the  diagonals.  This  increases
     parallelism substantially.
10 SCREEN 0,1,0
20 WIDTH 40
30 KEY OFF
40 COLOR 7,0,1
50 CLS
60 DIM XPOS(36),YPOS(36),NBR(36,7),COLR(36),CH%(36)
70 'read x and y positions for printing #s on screen
80 NODES=36
90 NZMAX=7
100 FOR I=1 TO NODES
110 READ XPOS(I)
120 NEXT
130 FOR I=1 TO NODES
140 READ YPOS(I)
150 NEXT
160 'read connection information into nbr array
170 FOR I=1 TO NODES
180 READ NZ
190 FOR J=1 TO NZ
200 READ NBR(I,J)
210 NEXT J
220 NBR(I,NZ+1)=0
230 NEXT I
240 'Set colors to initial values
250 FOR I=0 TO NODES
260 COLR(I)=0
270 NEXT I
280 ' Find # iterations desired
290 INPUT "# Iterations";ITERS
300 STEPCT=0
310 ' Find # steps desired
320 LOCATE 25,1
330 INPUT "# steps";STEPS
340 CLS
350 FOR STP=1 TO STEPS
360 FOR NODE=1 TO NODES
370 IF COLR(NODE)=ITERS GOTO 450
```

```
380 CH%(NODE)=1
390 FOR I=1 TO NZMAX
400 IF NBR(NODE,I)=0 GOTO 460
410 IF NBR(NODE,I)<NODE AND COLR(NBR(NODE,I))<
     COLR(NODE)+1 GOTO 450
420 IF NBR(NODE,I)>NODE AND COLR(NBR(NODE,I))<
     COLR(NODE) GOTO 450
430 NEXT I
440 GOTO 460
450 CH%(NODE)=0
460 NEXT NODE
470 ' Calculate new colors
480 FOR I=1 TO NODES
490 IF CH%(I) THEN COLR(I)=COLR(I)+1
500 NEXT I
510 ' Print node numbers in new colors
520 FOR NODE=1 TO NODES
530 LOCATE XPOS(NODE),YPOS(NODE)
540 COLOR (COLR(NODE) MOD 8) + 8
550 PRINT NODE;
560 NEXT NODE
570 ' Count number of nodes that changed
580 CHCT=0
590 FOR I=1 TO NODES
600 IF CH%(I) THEN CHCT=CHCT+1
610 NEXT I
620 LOCATE 21,20:COLOR 7
630 PRINT CHCT;"Nodes Updated "
640 ' End of step
650 STEPCT=STEPCT+1
660 FOR K=NODES TO 1 STEP -1
670 IF COLR(K)<ITERS THEN 700
680 NEXT K
690 GOTO 720
700 NEXT STP
710 GOTO 310
720 ' Done iteration. Print count of steps.
730 LOCATE 23,1:COLOR 7
740 PRINT STEPCT;"steps needed for";ITERS;"iterations";
750 ' Data values
800 ' Y Positions
805 DATA 5,5,7,9,7,5,5,7,9
810 DATA 11,13,11,9,7,5,5,7,9
815 DATA 11,13,15,15,13,11,9,7,9
820 DATA 11,13,15,15,13,11,13,15,15
825 ' X positions
830 DATA 5,10,5,5,10,15,20,15,10
835 DATA 4,4,9,14,19,24,29,24,19
840 DATA 14,9,4,9,14,19,24,29,29
845 DATA 24,19,14,19,24,29,29,24,29
1000 ' Connections. Format nz,nbr1,nbr2,...,nbrnz
```

```
1001 DATA 3,2,3,5
1002 DATA 4,1,5,6,8
1003 DATA 4,1,4,5,9
1004 DATA 4,3,9,10,12
1005 DATA 6,1,2,3,8,9,13
1006 DATA 4,2,7,8,14
1007 DATA 4,6,14,15,17
1008 DATA 6,2,5,6,13,14,18
1009 DATA 6,3,4,5,12,13,19
1010 DATA 4,4,11,12,20
1011 DATA 4,10,20,21,22
1012 DATA 6,4,9,10,19,20,23
1013 DATA 6,5,8,9,18,19,24
1014 DATA 6,6,7,8,15,17,18
1015 DATA 4,7,16,17,26
1016 DATA 2,15,26
1017 DATA 6,7,14,15,25,26,27
1018 DATA 6,8,13,14,24,25,28
1019 DATA 6,9,12,13,23,24,29
1020 DATA 6,10,11,12,22,23,30
1021 DATA 2,11,22
1022 DATA 4,11,20,21,30
1023 DATA 6,12,19,20,29,30,31
1024 DATA 6,13,18,19,28,29,32
1025 DATA 6,14,17,18,27,28,33
1026 DATA 4,15,16,17,27
1027 DATA 4,17,25,26,33
1028 DATA 6,18,24,25,32,33,34
1029 DATA 6,19,23,24,31,32,35
1030 DATA 4,20,22,23,31
1031 DATA 4,23,29,30,35
1032 DATA 6,24,28,29,34,35,36
1033 DATA 4,25,27,28,34
1034 DATA 4,28,32,33,36
1035 DATA 4,29,31,32,36
1036 DATA 3,32,34,35
```

APPENDIX B - Listing of BASIC Demonstration Program 2

```
1 ' Program to demonstrate operation
    of the parallel SOR algorithm
2 ' with 1 PE per node. This example
    has 36 nodes, connected like
3 ' FEM triangles:  x - x - x
4 '                 | \ | \ |
5 '                 x - x - x
6 '                 | \ | \ |
7 '                 x - x - x   etc.
8 ' The numbering scheme is a multicolor ordering.
10 SCREEN 0,1,0
20 WIDTH 40
30 KEY OFF
40 COLOR 7,0,1
50 CLS
60 DIM XPOS(36),YPOS(36),NBR(36,7),COLR(36),CH%(36)
70 'read x and y positions for printing #s on screen
80 NODES=36
90 NZMAX=7
100 FOR I=1 TO NODES
110 READ XPOS(I)
120 NEXT
130 FOR I=1 TO NODES
140 READ YPOS(I)
150 NEXT
160 'read connection information into nbr array
170 FOR I=1 TO NODES
180 READ NZ
190 FOR J=1 TO NZ
200 READ NBR(I,J)
210 NEXT J
220 NBR(I,NZ+1)=0
230 NEXT I
240 'Set colors to initial values
250 FOR I=0 TO NODES
260 COLR(I)=0
270 NEXT I
280 ' Find # iterations desired
290 INPUT "# Iterations";ITERS
300 STEPCT=0
310 ' Find # steps desired
320 LOCATE 25,1
330 INPUT "# steps";STEPS
340 CLS
350 FOR STP=1 TO STEPS
360 FOR NODE=1 TO NODES
370 IF COLR(NODE)=ITERS GOTO 450
380 CH%(NODE)=1
390 FOR I=1 TO NZMAX
```

```
400 IF NBR(NODE,I)=0 GOTO 460
410 IF NBR(NODE,I)<NODE AND COLR(NBR(NODE,I))<
     COLR(NODE)+1 GOTO 450
420 IF NBR(NODE,I)>NODE AND COLR(NBR(NODE,I))<
     COLR(NODE) GOTO 450
430 NEXT I
440 GOTO 460
450 CH%(NODE)=0
460 NEXT NODE
470 ' Calculate new colors
480 FOR I=1 TO NODES
490 IF CH%(I) THEN COLR(I)=COLR(I)+1
500 NEXT I
510 ' Print node numbers in new colors
520 FOR NODE=1 TO NODES
530 LOCATE XPOS(NODE),YPOS(NODE)
540 COLOR (COLR(NODE) MOD 8) + 8
550 PRINT NODE;
560 NEXT NODE
570 ' Count number of nodes that changed
580 CHCT=0
590 FOR I=1 TO NODES
600 IF CH%(I) THEN CHCT=CHCT+1
610 NEXT I
620 LOCATE 21,20:COLOR 7
630 PRINT CHCT;"Nodes Updated "
640 ' End of step
650 STEPCT=STEPCT+1
660 FOR K=NODES TO 1 STEP -1
670 IF COLR(K)<ITERS THEN 700
680 NEXT K
690 GOTO 720
700 NEXT STP
710 GOTO 310
720 ' Done iteration. Print count of steps.
730 LOCATE 23,1:COLOR 7
740 PRINT STEPCT;"steps needed for";ITERS;"iterations";
750 ' Data values
800 'Row locations for displaying node numbers on screen.
805 DATA 5,5,7,7,9,9,11,11,13
810 DATA 13,15,15,5,5,7,7,9,9
815 DATA 11,11,13,13,15,15,5,5,7
820 DATA 7,9,9,11,11,13,13,15,15
825 'Column locations for displaying node numbers.
830 DATA 5,20,15,30,10,25,5,20,15
835 DATA 29,9,24,9,24,4,19,14,29
840 DATA 9,24,4,19,14,29,14,29,9
845 DATA 24,4,19,14,29,9,24,4,19
1000 'Connections between nodes - nz,nbr1,nbr2,...,nbrnz
1001 DATA 3,13,15,27
1002 DATA 4,25,14,16,28
```

```
1003 DATA 6,13,25,27,16,17,30
1004 DATA 4,14,26,28,18
1005 DATA 6,15,27,29,17,19,31
1006 DATA 6,16,28,30,18,20,32
1007 DATA 4,29,21,19,33
1008 DATA 6,17,30,31,20,22,34
1009 DATA 6,19,31,33,22,23,36
1010 DATA 4,20,32,34,24
1011 DATA 4,21,33,35,23
1012 DATA 4,22,34,36,24
1013 DATA 4,1,25,27,3
1014 DATA 4,2,26,28,4
1015 DATA 4,1,27,29,5
1016 DATA 6,25,2,3,28,30,6
1017 DATA 6,27,3,5,30,31,8
1018 DATA 4,28,4,6,32
1019 DATA 6,29,5,7,31,33,9
1020 DATA 6,30,6,8,32,34,10
1021 DATA 4,7,33,35,11
1022 DATA 6,31,8,9,34,36,12
1023 DATA 4,33,9,11,36
1024 DATA 3,34,10,12
1025 DATA 4,13,2,3,16
1026 DATA 2,14,4
1027 DATA 6,1,3,15,3,5,17
1028 DATA 6,2,14,16,4,6,18
1029 DATA 4,15,5,7,19
1030 DATA 6,3,16,17,6,8,20
1031 DATA 6,5,17,19,8,9,22
1032 DATA 4,6,18,20,10
1033 DATA 6,7,19,21,9,11,23
1034 DATA 6,8,20,22,10,12,24
1035 DATA 2,21,11
1036 DATA 4,9,22,23,12
```

APPENDIX C - Simulation Program Listing

```
! This is a program to simulate the activities of a parallel array
  of processors, solving a system of linear equations by SOR
  iteration.  Each processor consists of an arithmetic unit and
  four communications processors, which communicate over external
  busses to other PEs and over internal busses with each other.
  Each bus has a passive class which maintains status.  Each CP
  also has a class, which contains registers and register status
  indicators.

      The PEs are numbered by row and column address in a grid.
  Within a PE, the CPs are numbered as follows:
      1 = Up       3 = Left
      2 = Down     4 = Right
  The external busses are numbered by the row and column of the
  PE which is the bus master (Down or Right), with an additional
  digit to indicate Down (0) or Right (1).;

Simulation BEGIN

    ! Declare global variables;
    INTEGER gi,gj,gk,gl,gm,gn,numnodes,numrows,numcols,iters,
    mdind,buscycl,prtlvl;
    INTEGER ARRAY nodeno[1:6,1:6],iter[1:36];
    INTEGER ARRAY needs[1:36,1:10],numnbrs[1:36],nbrsdone[1:36];
    INTEGER ARRAY destct[1:6,1:6,1:4],sdest[1:6,1:6,1:4,1:2],mdest[2:30];
    BOOLEAN ARRAY gotval[1:36,1:10];
    REAL tlim;
    REAL ARRAY t[1:10];
    REF(extbus) ARRAY ebus[1:6,1:6,0:1];
    REF(intbus) ARRAY ibus[1:6,1:6];
    REF(cpregs) ARRAY cpreg[1:6,1:6,1:4];
    REF(Head) ARRAY fifo[1:6,1:6,1:4];
    REF(aus) ARRAY au[1:36];
    REF(sendrslt) ARRAY sendres[1:6,1:6,1:4];
    REAL anscyc,assocyc,aucyc,multdel,muxcyc,rcdel,mutil;
! The following class contains the status and contents of the registers
  in a single CP. It will be used as
      Ref(cpregs) cpreg[row, col, cpnum];

    CLASS cpregs;
    BEGIN
        BOOLEAN auful,rtful,inful,auisans,
        nmful,rcudone,amemdone;
        BOOLEAN ibful;
        INTEGER auj,aur,auc,rtj,rtr,rtc,inj,inr,inc,rtdest,nmr,nmc;
        INTEGER ibj,ibr,ibc;
      auful:=FALSE;rtful:=FALSE;inful:=FALSE;
      ibful:=FALSE;
   END  --- cpregs;


! The following class contains the status of the external bus.
  It will be used as
     Ref(extbus) ebus[row, col, dir]
  where dir = 0 for down and 1 for right;

   CLASS extbus;
   BEGIN
      BOOLEAN req,grant,datrdy,bufful;
      req:=FALSE;grant:=FALSE;datrdy:=FALSE;bufful:=FALSE
   END  --- extbus;


! This class controls the status and allocation
  for an internal bus.;

   Process CLASS intbus(row,col);
   INTEGER row,col;
   BEGIN
      BOOLEAN datrdy,bufful,ack;
      BOOLEAN ARRAY req[1:4],creq[1:4];
      INTEGER grant,i,next,addr;
      FOR i:=1 STEP 1 UNTIL 4 DO
      req[i]:=FALSE;
      grant:=0;datrdy:=FALSE;
      bufful:=FALSE;
      next:=1;

      phase1:
      FOR i:=1 STEP 1 UNTIL 4 DO
      creq[i]:=req[i];
      Hold(.5);

      phase2:
      FOR i:=1 STEP 1 UNTIL 4 DO
      BEGIN
         IF creq[next] THEN GOTO cycle;
         next:=next+1;
         IF next>4 THEN next:=1
      END;
      GOTO done;

      cycle:
      grant:=next;
      IF prtlvl>2 THEN
```

```
    BEGIN
       Outtext("Int Bus for node");Outint(nodeno[row,col],3);
       Outtext(" granted to CP");Outint(next,3);
       Outfix(Time,1,8);Outimage
    END;
    Hold(.5);
    WHILE req[next] DO
    Hold(1);
    Hold(.5);
    IF prtlvl>2 THEN
    BEGIN
       Outtext("Int bus cleared, node");Outint(nodeno[row,col],3);
       Outfix(Time,1,8);Outimage
    END;
    next:=next+1;
    IF next>4 THEN next:=1;
    grant:=0;

    done:
    Hold(.5);
    GOTO phase1

 END --- intbus;


! The following defines the messages which will be stored in the
  FIFO buffer for sending off chip. They will be used as
   new mesg(j,r,c).into(fifo[row,col,cp]);

 Link CLASS mesg(j,r,c);
 INTEGER j,r,c;
 BEGIN
    INTEGER mj,mr,mc;
    mj:=j;mr:=r;mc:=c
 END --- mesg;


! The following process is the bus master for external data busses.
  This will only be used for Down and Right CPs.;

 Process CLASS ebusmast(row,col,cp);
 ! ***************************************;

 INTEGER row,col,cp;
 BEGIN
    INTEGER myrow,mycol,mycp;
    ! cvar used to record current state of var;
    BOOLEAN creq,cgrant,cdatrdy,cbufful;
    BOOLEAN cinful,businuse,latarm,cinfifo;
```

```
        INTEGER i,j,k,slicect,ebnum;
        INTEGER rcvrow,rcvcol,rcvcp;
        REF(mesg) mm;

        ! If this bus driver is on edge and unused, get rid of it;
        IF (cp=2 AND row=numrows) OR (cp=4 AND col=numcols) THEN Passivate;
        ! Remember own id;
        myrow:=row;mycol:=col;mycp:=cp;

        ! Initialize the variables which are not set before use;
        slicect:=0;
        businuse:=FALSE;latarm:=FALSE;

        ! Compute external bus name based on my type;
        IF mycp=2 THEN ebnum:=0 ELSE ebnum:=1;

        ! Compute address of slave bus driver which receives data from me;
        IF mycp=2 THEN
        BEGIN
           rcvrow:=myrow+1;rcvcol:=mycol;rcvcp:=1
        END
        ELSE
        BEGIN
           rcvrow:=myrow;rcvcol:=mycol+1;rcvcp:=3
        END;


! There are two phases to the processing:
    1 - read in the status information that affects my action,
    2 - do operations and change output statuses.
  There is a .5 clock cycle pause in between these so that all the
  processes which are affected by any status read the status before any
  have a chance to change it. This prevents race conditions. These are
  called phase1 and phase2;

        phase1:

        INSPECT ebus[myrow,mycol,ebnum] DO
        BEGIN
           creq:=req;
           cgrant:=grant;
           cdatrdy:=datrdy;
           cbufful:=bufful
        END;
        cinful:=cpreg[myrow,mycol,mycp].inful;
        cinfifo:= NOT(fifo[myrow,mycol,mycp].Empty);

        ! Phase 1 done. Wait before changing any status;
        Hold(.5);

        phase2:

! This is where things happen. First see if an operation is already
  in progress;

        IF businuse THEN GOTO sending;
        IF cgrant THEN GOTO receive;


        ! Bus is idle. See if it will stay idle;
        IF (cbufful AND cinful) OR (cinful AND NOT(cinfifo))
        OR (cbufful AND NOT(creq)) OR (NOT(cinfifo) AND NOT(creq))
        THEN GOTO done;

        ! Something is going to start happenning. See what;

        ! If slave wants bus and master doesn't, set "grant";
        IF creq AND (cbufful OR NOT(cinfifo)) AND NOT(cinful) THEN
        BEGIN
           ebus[myrow,mycol,ebnum].grant := TRUE;
           GOTO done
        END

        ! If master wants bus and slave buffer not full, master gets it;
        IF cinfifo AND NOT (cbufful) THEN
        BEGIN
           businuse:=TRUE;
           GOTO done
        END

        ! If processing ever gets to here, something is wrong;
        Outtext("Error in external bus operation at CP #");
        Outint(myrow,4);Outint(mycol,4);Outint(mycp,4);
        Outimage;
        GOTO done;

        ! Bus is in use to send data to slave;

        sending:

        IF cdatrdy THEN
        BEGIN
           ! In the middle of a bus cycle, so finish it;
           ebus[myrow,mycol,ebnum].datrdy:=FALSE;
           slicect:=slicect+1;
           GOTO done
        END;
```

```
    IF slicect<buscycl THEN
    BEGIN
       ! Have sent part of a word, so send the next part;
       ebus[myrow,mycol,ebnum].datrdy := TRUE;
       GOTO done
    END;

! All bus cycles needed to send a word are finished. Place
  actual data in receiver register, remove message from FIFO,
  clear up bus;
    mm:-fifo[myrow,mycol,mycp].First;
    cpreg[rcvrow,rcvcol,rcvcp].inj:=mm.mj;
    cpreg[rcvrow,rcvcol,rcvcp].inr:=mm.mr;
    cpreg[rcvrow,rcvcol,rcvcp].inc:=mm.mc;
    fifo[myrow,mycol,mycp].First.Out;

    businuse:=FALSE;
    slicect:=0;
    GOTO done;


    ! Bus is being used to receive data from slave;

    receive:

    IF slicect=buscycl THEN
    BEGIN
       ! All parts of word have been received. Clean up;
       ebus[myrow,mycol,ebnum].grant:=FALSE;
       cpreg[myrow,mycol,mycp].inful:=TRUE;
       slicect:=0;
       ! The following should not be needed, but is included for safety;
       latarm:=FALSE;
       IF prtlvl>1 THEN
       BEGIN
          Outtext("EM Rcvd at node=");
          Outint(nodeno[row,col],3);Outtext("  CP=");
          Outint(cp,3);Outtext("  with j=");
          Outint(cpreg[row,col,cp].inj,3);
          Outfix(Time,1,6);Outimage
       END;
       GOTO done
    END;

    ! Not done with word, so get more in;
    IF cdatrdy THEN
    BEGIN
       ! Data Ready -> 1 says arm latch;
```

```
        latarm:=TRUE;
        GOTO done
    END;

    IF latarm THEN
    BEGIN
! At this point, know that Data Ready = 0. If
  it was previously 1, latch is armed, so will
  get data;
        latarm:=FALSE;
        slicect:=slicect+1
    END;

    ! Finished this clock cycle. Wait for start of next;
    done:

    Hold(.5);
    GOTO phase1
  END --- ebusmast;

! The following process is the bus slave for external data busses.
  This will only be used for Up and Left CPs.;

  Process CLASS ebusslav(row,col,cp);
  ! ***************************************;

  INTEGER row,col,cp;
  BEGIN
      INTEGER myrow,mycol,mycp;
      ! cvar used to record current state of var;
      BOOLEAN creq,cgrant,cdatrdy,cbufful;
      BOOLEAN cinful,latarm,cinfifo;
      INTEGER i,j,k,slicect,ebnum;
      INTEGER rcvrow,rcvcol,rcvcp;
      REF(mesg) mm;

      ! If this bus driver is on the edge of the array, delete it;
      IF (cp=1 AND row=1) OR (cp=3 AND col=1) THEN Passivate;

      ! Remember own id;
      myrow:=row;mycol:=col;mycp:=cp;

      ! Initialize the variables which are not set before use;
      slicect:=0;
      latarm:=FALSE;

      ! Compute external bus name based on my type;
      IF mycp=1 THEN ebnum:=0 ELSE ebnum:=1;
      ! Compute address of master bus driver which receives data from me;
      IF mycp=1 THEN
      BEGIN
         rcvrow:=myrow-1;rcvcol:=mycol;rcvcp:=2
      END
      ELSE
      BEGIN
         rcvrow:=myrow;rcvcol:=mycol-1;rcvcp:=4
      END;


! There are two phases to the processing:
    1 - read in the status information that affects my action,
    2 - do operations and change output statuses.
  There is a .5 clock cycle pause in between these so that all the
  processes which are affected by any status read the status before any
  have a chance to change it.  This prevents race conditions.  These are
  called phase1 and phase2;

      phase1:

      INSPECT ebus[rcvrow,rcvcol,ebnum] DO
      BEGIN
         creq:=req;
         cgrant:=grant;
         cdatrdy:=datrdy;
         cbufful:=bufful
      END;
      cinfifo:= NOT(fifo[myrow,mycol,mycp].Empty);

      ! Phase 1 done.  Wait before changing any status;
      Hold(.5);

      phase2:

! This is where things happen. First see if an operation is already
  in progress;

      IF cgrant THEN GOTO sending;
      IF cinfifo THEN ebus[rcvrow,rcvcol,ebnum].req:=TRUE;
      IF latarm THEN GOTO receive;
      IF cdatrdy THEN latarm:=TRUE;
      GOTO done;

      receive:
      ! I am in the process of receiving data. Continue;

      IF NOT(cdatrdy) THEN
      BEGIN
    slicect:=slicect+1;
    latarm:=FALSE;
    IF slicect=buscycl THEN
    BEGIN
       IF prtlvl>1 THEN
       BEGIN
          Outtext("ES Rcvd at node=");
          Outint(nodeno[row,col],3);Outtext("  CP=");
          Outint(cp,3);Outtext("  with j=");
          Outint(cpreg[row,col,cp].inj,3);
          Outfix(Time,1,6);Outimage
       END;
       cpreg[myrow,mycol,mycp].inful:=TRUE;
       slicect:=0;
       ebus[rcvrow,rcvcol,ebnum].bufful:=TRUE
    END
END;
GOTO done;

sending:
! I am sending data. Continue;

IF cdatrdy THEN
BEGIN
    ebus[rcvrow,rcvcol,ebnum].datrdy:=FALSE;
    slicect:=slicect+1;
    GOTO done
END;

IF slicect=buscycl THEN
BEGIN
    ebus[rcvrow,rcvcol,ebnum].req:=FALSE;
    mm:-fifo[myrow,mycol,mycp].First;
    slicect:=0;
    cpreg[rcvrow,rcvcol,rcvcp].inj:=mm.mj;
    cpreg[rcvrow,rcvcol,rcvcp].inr:=mm.mr;
    cpreg[rcvrow,rcvcol,rcvcp].inc:=mm.mc;
    fifo[myrow,mycol,mycp].First.Out;
    GOTO done
END;

ebus[rcvrow,rcvcol,ebnum].datrdy:=TRUE;

done:

! Finished this clock cycle. Wait for start of next;

Hold(.5);
GOTO phase1
```

```
      END  --- ebusslav;

      Process CLASS ibusdriv(row,col,cp);
      ! ********************************************;
      INTEGER row,col,cp;
      BEGIN
         INTEGER cgrant,caddr,dest;
         BOOLEAN cdatrdy,cack,crtful,cibful,receive,
         latarm,requested,waitack,waitclr,waitdat;

         phase1:
         INSPECT ibus[row,col] DO
         BEGIN
            cgrant:=grant;
            caddr:=addr;
            cdatrdy:=datrdy;
            cack:=ack;
            crtful:=cpreg[row,col,cp].rtful;
            cibful:=cpreg[row,col,cp].ibful
         END;
         Hold(.5);

         phase2:

         ! If activity is already in process, work on it;
         IF waitack THEN GOTO sending;
         IF waitclr THEN GOTO rcving;
         IF waitdat THEN GOTO getdata;

   ! Nothing in progress. See if something will start.
     First, set request if I have data to send;
         IF NOT(requested) AND crtful THEN
         BEGIN
            ibus[row,col].req[cp]:=TRUE;
            requested:=TRUE;
            GOTO listen
         END;

         ! See if I got the bus for my use;
         IF requested AND cgrant=cp THEN
         BEGIN
            ! I have been granted access to bus, so send;
            Hold(1);
            ibus[row,col].datrdy:=TRUE;
            dest:=cpreg[row,col,cp].rtdest;
            ibus[row,col].addr:=dest;
            waitack:=TRUE;
            GOTO done
```

```
END;

listen:
! I don't have the bus. See if someone sending to me;
IF caddr=cp THEN
BEGIN
   waitclr:=TRUE;
   IF cdatrdy THEN latarm:=TRUE ELSE latarm:=FALSE
END;
GOTO done;


sending:
IF cack THEN
BEGIN
   ! Below are steps needed to send data;
   Hold(1);
   ibus[row,col].datrdy:=FALSE;
   Hold(1);
   cpreg[row,col,dest].ibr:=
   cpreg[row,col,cp].rtr;
   cpreg[row,col,dest].ibc:=
   cpreg[row,col,cp].rtc;
   cpreg[row,col,dest].ibj:=
   cpreg[row,col,cp].rtj;
   cpreg[row,col,cp].rtful:=FALSE;
   ibus[row,col].req[cp]:=FALSE;
   ibus[row,col].addr:=0;
   waitack:=FALSE;
   IF prtlvl>2 THEN
   BEGIN
      Outtext("IB driver sent data from node");Outint(nodeno[row,col],3);
      Outtext(" CP");Outint(cp,3);Outtext(" to CP");Outint(dest,3);
      Outtext(" j=");Outint(cpreg[row,col,cp].rtj,3);Outfix(Time,1,8);
      Outimage
   END;
   requested:=FALSE;
   ! Wait for bus to clear;
   Hold(1)
END;
! Finished or no "ACK" received yet;
GOTO done;

rcving:
IF latarm THEN
BEGIN
   IF cibful THEN GOTO done;
   ibus[row,col].ack:=TRUE;
   waitclr:=FALSE;
```

```
            waitdat:=TRUE;
            GOTO done
        END
        ELSE IF cdatrdy THEN latarm:=TRUE;
        GOTO done;

        getdata:
        IF cdatrdy THEN GOTO done;
        ! Data Ready dropped, so latch in data;
        waitdat:=FALSE;
        latarm:=FALSE;
        cpreg[row,col,cp].ibful:=TRUE;
        IF prtlvl>2 THEN
        BEGIN
            Outtext("IB Data Rcvd, node");Outint(nodeno[row,col],3);
            Outtext(" CP");Outint(cp,3);Outfix(Time,1,8);
            Outimage
        END;
        Hold(1);
        ibus[row,col].ack:=FALSE;

        done:
        Hold(.5);
        GOTO phase1
    END --- ibusdriv;

    Process CLASS assocmem(row,col,cp,i);
    ! ****************************************;

    INTEGER row,col,cp,i;
    BEGIN
        INTEGER j,k;
        BOOLEAN needed,cauful,cinful;

! This process accesses a global array, needs[i,j], which
  stores the needed node number values for each node as
  needs[i,1]=j1, needs[i,2]=j2, ..., needs[i,n]=jn,
  needs[i,n+1]=0. It works in two phases, like the other
  processes;

        phase1:

        INSPECT cpreg[row,col,cp] DO
        BEGIN
            cauful:=auful;
            cinful:=inful
        END;

        Hold(.5);
        phase2:

        IF cauful OR  NOT(cinful) THEN GOTO done;
   ! There is data to be processed in the input register. Global
     variable assocyc contains the number of clock cycles which
     the associate memory is assumed to require to finish its job.;

   ! First, the asociative memory determines whether the data item is
     needed at this location;
        j:=cpreg[row,col,cp].inj;
        k:=1;
        needed:=FALSE;
        loop:
        IF needs[i,k]=j AND NOT(gotval[i,k]) THEN
        BEGIN
           needed:=TRUE;
           gotval[i,k]:=TRUE
        END
        ELSE IF needs[i,k] \= 0 THEN
        BEGIN
           k:=k+1;
           GOTO loop
        END;

        ! Simulate the processing time;
        Hold(assocyc);

        ! Update status of the system;
        IF needed THEN
        cpreg[row,col,cp].auful:=TRUE;

        ! Signal that associative memory unit is finished with input data;
        cpreg[row,col,cp].amemdone:=TRUE;

        done:
        ! Finish this cycle;
        Hold(.5);
        GOTO phase1
     END  --- assocmem;


     Process CLASS aus(row,col,i);
     ! **********************************;

     INTEGER row,col,i;
     BEGIN
        INTEGER next,numpnd,j,k,mbusy,midle;
        BOOLEAN mulinprog,cauful;
```

```
      next:=1;
      numpnd:=0;
      mulinprog:=FALSE;
      mbusy:=midle:=0;

      phase1:

      IF mulinprog THEN mbusy:=mbusy+1 ELSE midle:=midle+1;
      cauful:=cpreg[row,col,next].auful AND
      NOT(cpreg[row,col,next].auisans);
      Hold(.5);

      phase2:

      IF cauful THEN
      BEGIN
         Hold(aucyc);
         cpreg[row,col,next].auful:=FALSE;
         numpnd:=numpnd+1
      END;

      next:=next+1;
      IF next>4 THEN next:=1;

      IF (numpnd>0 AND NOT(mulinprog)) THEN
      BEGIN
         IF prtlvl>1 THEN
         BEGIN
            Outtext("Mult started at node");Outint(i,3);
            Outfix(Time,1,6);Outimage
         END;
         mulinprog:=TRUE;
         numpnd:=numpnd-1;
         ACTIVATE NEW mults(i) DELAY multdel
      END;

      IF nbrsdone[i]=numnbrs[i] THEN
      BEGIN
! Clear array "gotval" so the assocmem will look again at
  the needed values for the next iteration;
         k:=0;
         FOR k:=k+1 WHILE needs[i,k] \=0 DO
         gotval[i,k]:=FALSE;
         iter[i]:=iter[i]+1;
         IF prtlvl>0 THEN
         BEGIN
            Outtext("Ans ready at node");Outint(i,4);
            Outtext(" for iteration number");Outint(iter[i],3);
            Outfix(Time,1,8);Outimage
         END;
! Keep track of time needed to do iterations and stop when requested
  iterations are done. Do this only for last node;
         IF i=numnodes THEN
         BEGIN
            t[iter[i]]:=Time;
            IF iter[i] >= iters  THEN REACTIVATE Main
         END;
         Hold(anscyc);
         FOR k:=1 STEP 1 UNTIL 4 DO
         BEGIN
            cpreg[row,col,k].auful:=TRUE;
            cpreg[row,col,k].auisans:=TRUE;
            ACTIVATE sendres[row,col,k] DELAY .5
         END;

! After first update at a node, set nbrsdone to -1, and
  numpnd to 1, so multiplier computes (1-w)*x(i);
         numpnd:=1;
         nbrsdone[i]:=-1
      END  --- if;

      Hold(.5);
      GOTO phase1
   END  --- aus;


   Process CLASS mults(i);
   ! *****************************;

   INTEGER i;
   BEGIN
! This process is unnamed.  No one else refers to it.  All it
  does is tell the AU process of the same number that a
  certain amount of time has elapsed, equal to the time
  needed for a multiplication.;
      au[i].mulinprog:=FALSE;
      nbrsdone[i]:=nbrsdone[i]+1
   END  mults;

   Process CLASS rcupdt(row,col,cp);
   ! **************************************;

   INTEGER row,col,cp;
   BEGIN
      BOOLEAN cinful,crtful;
      INTEGER r,c,dest;
```

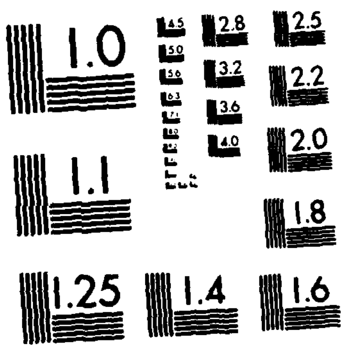
```
        phase1:
        INSPECT cpreg[row,col,cp] DO
        BEGIN
           cinful:=inful;
           crtful:=rtful
        END;

        Hold(.5);

        phase2:

        INSPECT cpreg[row,col,cp] DO
        BEGIN
           r:=inr;
           c:=inc
        END;

        IF (r=0 AND c=0 AND cinful) THEN
        BEGIN
           Hold(1);
           cpreg[row,col,cp].rcudone:=TRUE;
           GOTO done
        END;

        IF NOT(crtful) AND cinful THEN
        BEGIN
           Hold(rcdel);

! Processing of r and c depends on which CP this is.
  1=Up, 2=Down, 3=Left, 4=Right.  Page 47 in notes
  describes logic for this.;

           IF cp=1 THEN
           BEGIN
              ! For Up, r>0 => send to down;
              IF r>0 THEN
              BEGIN
                 r:=r-1;
                 dest:=2;
                 GOTO movregs
              END
           END

           ELSE IF cp=2 THEN
           BEGIN
! For Down, may have:
  r<0      => r++, send to Up (dest=1)
  r=0,c>0 => c--, send to Rt (dest=4);
              IF r<0 THEN
```

```
                BEGIN
                    r:=r+1;
                    dest:=1;
                    GOTO movregs
                END
                ELSE IF c>0 THEN
                BEGIN
                    c:=c-1;
                    dest:=4;
                    GOTO movregs
                END
            END

            ELSE IF cp=3 THEN
            BEGIN
! For Left, actions are:
   c>0      => c--, send to Rt (Dest=4)
   c=0,r>0 => r--, send to Dn (Dest=2);
                IF c>0 THEN
                BEGIN
                    c:=c-1;
                    dest:=4;
                    GOTO movregs
                END
                ELSE IF r>0 THEN
                BEGIN
                    r:=r-1;
                    dest:=2;
                    GOTO movregs
                END
            END

            ELSE IF cp=4 THEN
            BEGIN
! For Right, there are several cases:
   c<0      => c++, to Left (dest=3)
   c=0,r>0 => r--, to Down (dest=2)
   c=0,r<0 => r++, to Up   (dest=1);
                IF c<0 THEN
                BEGIN
                    c:=c+1;
                    dest:=3;
                    GOTO movregs
                END
                ELSE IF r>0 THEN
                BEGIN
                    r:=r-1;
                    dest:=2;
                    GOTO movregs
                END
                ELSE IF r<0 THEN
                BEGIN
                   r:=r+1;
                   dest:=1;
                   GOTO movregs
                END
             END
          END  --- of long if-then-ELSE;
          GOTO done;

          movregs:
          cpreg[row,col,cp].rtr:=r;
          cpreg[row,col,cp].rtc:=c;
          cpreg[row,col,cp].rtj:=cpreg[row,col,cp].inj;
          cpreg[row,col,cp].rtful:=TRUE;
          cpreg[row,col,cp].rcudone:=TRUE;
          cpreg[row,col,cp].rtdest:=dest;
          IF prtlvl>2 THEN
          BEGIN
             Outtext("RC Update, node");Outint(nodeno[row,col],3);
             Outtext(" CP");Outint(cp,3);Outtext(" to CP");
             Outint(dest,3);Outfix(Time,1,8);
             Outimage
          END;

          done:
          Hold(.5);
          GOTO phase1
       END  --- rcupdt;


       Process CLASS inempt(row,col,cp);
       ! *************************************;

       INTEGER row,col,cp;
       BEGIN
    ! This process checks whether both the associative memory
      and the rcupdate unit have finished with the data in the
      input buffer. If so, it empties the buffer.;
          BOOLEAN cinful,crcu,cam;
          INTEGER ebnum;
          IF cp=1 THEN ebnum:=0 ELSE ebnum:=1;

          phase1:
          INSPECT cpreg[row,col,cp] DO
          BEGIN
             cinful:=inful;
             crcu:=rcudone;
```

```
            cam:=amemdone;
            Hold(.5);

            phase2:
            IF cinful AND crcu AND cam THEN
            BEGIN
               inful:=rcudone:=amemdone:=FALSE;
               IF cp=1 THEN
               ebus[row-1,col,ebnum].bufful:=FALSE
               ELSE IF cp=3 THEN
               ebus[row,col-1,ebnum].bufful:=FALSE;
            END;
            Hold(.5);
            GOTO phase1
         END
      END --- inempt;


      Process CLASS sendrslt(row,col,cp);
      ! *****************************************;

      INTEGER row,col,cp;
      BEGIN
   ! This process takes results received from the AU and
     sends them to the mecessary receivers, placing the
     R & C values for the destination(s) on them;
         BOOLEAN cnmful;
         INTEGER dcount,indx,k,sendct;
         INTEGER ARRAY rdest[1:3],cdest[1:3];

   ! On the first activation of this process, it determines
     whether there is one or more than one destination for
     the message sent via this CP. Then it reads in the
     destination(s). This information comes from global
     arrays destct and sdest or mdest;

         dcount:=destct[row,col,cp];
         IF dcount=1 THEN
         BEGIN
            rdest[1]:=sdest[row,col,cp,1];
            cdest[1]:=sdest[row,col,cp,2]
         END
         ELSE IF dcount>1 THEN
         BEGIN
            indx:=dcount;
            dcount:=mdest[indx];
            FOR k:=1 STEP 1 UNTIL dcount DO
            rdest[k]:=mdest[indx+k];
            indx:=indx+dcount;
            FOR k:=1 STEP 1 UNTIL dcount DO
            cdest[k]:=mdest[indx+k]
         END;
         sendct:=0;
! Wait til activated for a particular case. Note that this
  process is activated only when there is something for it
  to do, unlike many others.;
         Passivate;

         phase1:

         cnmful:=cpreg[row,col,cp].nmful;
         Hold(.5);

         phase2:
! The AU buffer contains the result of the latest
  computation in the AU. Forward it to the rest
  of the network as needed.;

         ! If the answer doesn't go anywhere, get it out of buffer;
         IF dcount=0 THEN
         BEGIN
            cpreg[row,col,cp].auful:=FALSE;
            cpreg[row,col,cp].auisans:=FALSE;
            Passivate;
            GOTO phase1
         END;

         ! First, make sure the new message buffer is available.;
         IF cnmful THEN
         BEGIN
            Hold(.5);
            GOTO phase1
         END;

         ! Buffer is available. Send data to it;
         sendct:=sendct+1;
         INSPECT cpreg[row,col,cp] DO
         BEGIN
            nmful:=TRUE;
            nmr:=rdest[sendct];
            nmc:=cdest[sendct]
         END;
         IF prtlvl>2 THEN
         BEGIN
            Outtext("Message ");Outint(sendct,3);Outtext(" of");
            Outint(dcount,3);Outtext(" sent from node");
            Outint(nodeno[row,col],3);
            Outtext(", CP");Outint(cp,3);Outfix(Time,1,8);Outimage
```

```
        END;
        ! See if all messages have been sent;
        IF sendct=dcount THEN
        BEGIN
            sendct:=0;
            cpreg[row,col,cp].auful:=FALSE;
            cpreg[row,col,cp].auisans:=FALSE;
            Passivate
        END;

        Hold(.5);
        GOTO phase1
    END  --- sendrslt;


    Process CLASS fifomux(row,col,cp,i);
    ! *******************************************;

    INTEGER row,col,cp,i;
    BEGIN
! This process selects a data source from among several
  to place data into the output queue.;

        INTEGER k,nr,nc,ir,ic,j;
        BOOLEAN cibful,cnmful;

        phase1:
        INSPECT cpreg[row,col,cp] DO
        BEGIN
            cibful:=ibful;
            cnmful:=nmful;
        END;
        Hold(.5);

        phase2:
! Priority goes to the New Message buffer, since it seldom
  contains data.  A round robin scheme is used to select from
  the two internal busses for those CPs which have two.;
        IF cnmful THEN
        BEGIN
            Hold(muxcyc);
            ! ****;
            WHILE NOT(fifo[row,col,cp].Empty) DO
            Hold(1);
            INSPECT cpreg[row,col,cp] DO
            BEGIN
                nr:=nmr;nc:=nmc
            END;
            NEW mesg(i,nr,nc).Into(fifo[row,col,cp]);
            cpreg[row,col,cp].nmful:=FALSE;
            GOTO done
        END;

        IF cibful THEN
        BEGIN
            ! ****;
            WHILE NOT(fifo[row,col,cp].Empty) DO
            Hold(1);
            INSPECT cpreg[row,col,cp] DO
            BEGIN
                ir:=ibr;
                ic:=ibc;
                j:=ibj
            END;
            NEW mesg(j,ir,ic).Into(fifo[row,col,cp]);
            cpreg[row,col,cp].ibful:=FALSE
        END;

        done:
        Hold(.5);
        GOTO phase1
    END  --- fifomux;

    ! Main program;
    ! ***************************************;

    ! Get values of parameters for simulation run;
    INSPECT NEW Infile ("parms.run") DO
    BEGIN
        Open(Blanks(80));
        numnodes:=Inint;numrows:=Inint;numcols:=Inint;
        anscyc:=Inreal;assocyc:=Inreal;aucyc:=Inreal;
        buscycl:=Inint;multdel:=Inreal;
        muxcyc:=Inreal;rcdel:=Inreal;iters:=Inint;tlim:=Inreal;
        prtlvl:=Inint;
        Close
    END;

    ! Initialize iteration counting array, iter[i];
    FOR gi:=1 STEP 1 UNTIL numnodes DO
    iter[gi]:=0;

    ! Read in arrays describing network interconnections;

! First read needs[i,j]. Also get numnbrs filled while
  doing this;
    INSPECT NEW Infile("needs.run") DO
    BEGIN
    Open(Blanks(80));
    gi:=0;
    loop1:
    gi:=gi+1;
    gj:=Inint;
    IF gj>0 THEN
    BEGIN
        FOR gk:=1 STEP 1 UNTIL gj DO
        needs[gi,gk]:=Inint;
        needs[gi,gj+1]:=0;
        numnbrs[gi]:=gj;
        GOTO loop1
    END;
    Close
END;

! Next, read destinations for messages from each CP;
mdind:=2;
INSPECT NEW Infile("dests.run") DO
BEGIN
    Open(Blanks(80));
    FOR gi:=1 STEP 1 UNTIL numrows DO
    FOR gj:=1 STEP 1 UNTIL numcols DO
    FOR gk:=1 STEP 1 UNTIL 4 DO
    BEGIN
        destct[gi,gj,gk]:=Inint;
        IF destct[gi,gj,gk]=1 THEN
        BEGIN
            sdest[gi,gj,gk,1]:=Inint;
            sdest[gi,gj,gk,2]:=Inint
        END
        ELSE IF destct[gi,gj,gk]>0 THEN
        BEGIN
            gn:=destct[gi,gj,gk];
            destct[gi,gj,gk]:=mdind;
            mdest[mdind]:=gn;
            mdind:=mdind+1;
            gl:=2*gn;
            FOR gm:=1 STEP 1 UNTIL gl DO
            BEGIN
                ! Read all row destinations, then columns;
                mdest[mdind]:=Inint;
                mdind:=mdind+1
            END
        END
    END triple for loop;
    Close
END  infile;
```

AD-A171 256   A PARALLEL PROCESSOR FOR THE SOLUTION OF LARGE SPARSE   4/4
SYMMETRIC LINEAR SYSTEMS(U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH   D A ARPIN 1986
UNCLASSIFIED   AFIT/CI/NR-86-124D   F/G 9/2   NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

```
! Compute initial values for nbrsdone by subtracting 1 from
  numnbrs for each lower number node in needs;
    FOR gi:=1 STEP 1 UNTIL numnodes DO
    BEGIN
        gl:=numnbrs[gi];
        nbrsdone[gi]:=gl;
        FOR gj:=1 STEP 1 UNTIL gl DO
        IF needs[gi,gj]<gi THEN
        BEGIN
            nbrsdone[gi]:=nbrsdone[gi]-1;
            gotval[gi,gj]:=FALSE
        END
        ELSE gotval[gi,gj]:=TRUE
    END;

! Read in array of assignments of node numbers to
  row and column positions;
    INSPECT NEW Infile ("nodeno.run") DO
    BEGIN
        Open(Blanks(80));
        FOR gi:=1 STEP 1 UNTIL numrows DO
        FOR gj:=1 STEP 1 UNTIL numcols DO
        nodeno[gi,gj]:=Inint;
        Close
    END;

! All arrays are now set. Initialize processes and classes
  which implement the model;

    FOR gi:=1 STEP 1 UNTIL numrows DO
    FOR gj:=1 STEP 1 UNTIL numcols DO
    BEGIN
        ebus[gi,gj,0]:-NEW extbus;
        ebus[gi,gj,1]:-NEW extbus;
        FOR gk:=1 STEP 1 UNTIL 3 DO
        ibus[gi,gj]:-NEW intbus(gi,gj);
        ACTIVATE ibus[gi,gj] DELAY 0;
        gn:=nodeno[gi,gj];
        au[gn]:-NEW aus(gi,gj,gn);
        ACTIVATE au[gn] DELAY 0;

        FOR gk:=1 STEP 1 UNTIL 4 DO
        BEGIN
            ACTIVATE NEW ibusdriv(gi,gj,gk) DELAY 0;
            cpreg[gi,gj,gk]:-NEW cpregs;
            IF gk=2 OR gk=4 THEN
            ACTIVATE NEW ebusmast(gi,gj,gk) DELAY 0
            ELSE ACTIVATE NEW ebusslav(gi,gj,gk) DELAY 0;
            ACTIVATE NEW assocmem(gi,gj,gk,gn) DELAY 0;
```

```
          ACTIVATE NEW rcupdt(gi,gj,gk) DELAY 0;
          ACTIVATE NEW inempt(gi,gj,gk) DELAY 0;
          sendres[gi,gj,gk]:-NEW sendrslt(gi,gj,gk);
          ACTIVATE sendres[gi,gj,gk] DELAY 0;
          ACTIVATE NEW fifomux(gi,gj,gk,gn) DELAY 0;
          fifo[gi,gj,gk]:-NEW Head
       END
    END for;

    ! Print run parameters;
    Outtext("Array Simulated:");Outimage;
    FOR gi:=1 STEP 1 UNTIL numrows DO
    BEGIN
       FOR gj:=1 STEP 1 UNTIL numcols DO
       Outint(nodeno[gi,gj],5);
       Outimage
    END;
    Outtext("Multiplication takes");Outfix(multdel,1,6);
    Outtext(" clock cycles.");Outimage;
    Outtext("External bus transfers take");Outint(buscycl,3);
    Outtext(" clock cycles.");Outimage;
    FOR gi:=1 STEP 1 UNTIL numnodes DO
    BEGIN
       Outtext("Node");Outint(gi,3);
       Outtext(" is connected to nodes");
       gk:=numnbrs[gi];
       FOR gj:=1 STEP 1 UNTIL gk DO
       Outint(needs[gi,gj],3);
       Outimage
    END;
    Outint(iters,3);
    Outtext(" iterations will be run with a time limit of");
    Outfix(tlim,1,8);Outtext(" steps");Outimage;

    ! Let computing process work;
    Hold(tlim);

! At end of simulation, process au will reactivate this.
  Print out results;
    FOR gi:=1 STEP 1 UNTIL iters DO
    BEGIN
       Outtext("Iteration # ");Outint(gi,3);
       Outtext(" finished at time = ");Outfix(t[gi],1,8);
       Outtext(" clock cycles.");Outimage
    END
    ! Print out utilization for multiplier in each node;
    Outtext("Node #   Processor Busy Time (%)");Outimage;
    FOR gi:=1 STEP 1 UNTIL numnodes DO
    BEGIN
```

```
      mutil:=100*au[gi].mbusy/(au[gi].mbusy+au[gi].midle);
      Outint(gi,5);Outtext("                ");Outfix(mutil,2,8);
      Outimage
   END
END  simulation;
```

APPENDIX D - ABEL Listings for Prototype PALs

This section lists the inputs to the ABEL program which were used to construct the logic of the PAL chips used in the ParSOR prototype. In these listings, the # symbol indicates the logical OR operation, & indicates logical AND, and ! indicates NOT.

```
module EBMR flag '-r2'
title 'External Bus Master Register part, PAL 16R4
Dave Arpin and Bob Kaucic  Dec 3, 1985'
  uel   device   'P16R4';

"P16R4"
  Clock      pin in uel  1;
  _R         pin in uel  2;
  NMFul      pin in uel  3;
  IBBFul     pin in uel  4;
  INFul      pin in uel  5;
  Req        pin in uel  6;
  DataRdy_SM pin in uel  7;
  BUFFul     pin in uel  8;

"Outputs"
  DataRdy_MS pin in uel 19;
  _EBon      pin in uel 18;
  Q2         pin in uel 17;
  Q1         pin in uel 16;
  Q0         pin in uel 15;
  OBFul      pin in uel 14;
  _SNDNMB    pin in uel 13;
  _SNDIBB    pin in uel 12;

"Pin 11 will be grounded to enable outputs"

"States"
 S0=^b111;
 S1=^b110;
 S2=^b101;
 S3=^b100;
 S4=^b011;
 S5=^b010;
 S6=^b001;
 S7=^b000;

equations
ENABLE DataRdy_MS  = 1;
ENABLE _EBon       = 1;
ENABLE _SNDNMB      = 1;
ENABLE _SNDIBB      = 1;

STATE_DIAGRAM IN uel [Q2,Q1,Q0]

 state S0 : !_SNDNMB = !OBFul & NMFul;
            !_SNDIBB = !OBFul & !NMFul & IBBFul;
            OBFul    := OBFul;
            !_EBon    = OBFul & !BUFFul;
            DataRdy_MS = OBFul & !BUFFul;

            if !_R then S6
            else
            if !OBFul & NMFul then S1
            else
            if !OBFul & !NMFul & IBBFul then S2
            else
            if OBFul & !BUFFul then S3
            else
            if !INFul & Req & (!OBFul # BUFFul) then S4
            else S0;

state S1 : !_SNDNMB = 1;
           OBFul := 1;
           !_SNDIBB = 0;
           DataRdy_MS = 0;
           !_EBon = 0;
           if !_R then S6 else S0;

state S2 : !_SNDIBB = 1;
           OBFul := 1;
           !_SNDNMB = 0;
           DataRdy_MS = 0;
           !_EBon = 0;
           if !_R then S6 else S0;

state S3 : !_EBon = 1;
           DataRdy_MS = 1;
           OBFul := 0;
           !_SNDIBB = 0;
           !_SNDNMB = 0;
           if !_R then S6 else S0;

state S4 : OBFul := OBFul;
           !_SNDIBB = 0;
           !_SNDNMB = 0;
           DataRdy_MS = 0;
           !_EBon = 0;
           if !_R then S6
           else
           if DataRdy_SM then S5
```

```
              else S4;

state S5 : OBFul := OBFul;
           !_SNDIBB = 0;
           !_SNDNMB = 0;
           DataRdy_MS = 0;
           !_EBon = 0;
           if !_R then S6
           else
           if DataRdy_SM then S0
           else S5;

"Take care of power up conditions"

state S6 : OBFul := 0;
           !_SNDIBB = 0;
           !_SNDNMB = 0;
           !_EBon = 0;
           DataRdy_MS = 0;
           if !_R then S6 else S0;

state S7 : GOTO S6;

end EBMR;
```

```
module EBML flag '-r2'
title 'Written by Bob Kaucic'
 ue2    device   'P16L8';

"P16L8"
AUFull           pin in ue2  1;
AUOld            pin in ue2  2;
AMDone           pin in ue2  3;
_Needed_Here     pin in ue2  4;
RTFull           pin in ue2  5;
RTOld            pin in ue2  6;
RCUDone          pin in ue2  7;
Send_On          pin in ue2  8;
DataRdy_SM       pin in ue2  9;
Q0               pin in ue2 11;
Q1               pin in ue2 13;
Q2               pin in ue2 14;

_Clr_INFul       pin in ue2 12;
Set_INFul        pin in ue2 15;
Clock_OB         pin in ue2 16;
_Clr_NMBuff      pin in ue2 17;
_Clr_IBBuff      pin in ue2 18;
OB_Sel           pin in ue2 19;  "1 is upper bits"


equations in ue2
ENABLE OB_Sel      = 1;
ENABLE _Clr_IBBuff = 1;
ENABLE _Clr_NMBuff = 1;
ENABLE Clock_OB    = 1;
ENABLE Set_INFul   = 1;
ENABLE _Clr_INFul  = 1;

ENABLE Q2         = 0;
ENABLE Q1         = 0;

!_Clr_INFul          =   ((AUFull & !AUOld)
                           # (AMDone & _Needed_Here)) &
                          ((RTFull & !RTOld)
                           # (RCUDone & !Send_On))
                          # (!Q2 & !Q1 &  Q0);
 Set_INFul           =   (!Q2 &  Q1 & !Q0 & DataRdy_SM);
 Clock_OB            =  (( Q2 &  Q1 & !Q0) #
                          ( Q2 & !Q1 &  Q0));
!_Clr_NMBuff         =   ( Q2 &  Q1 & !Q0) #
                          (!Q2 & !Q1 &  Q0);
!_Clr_IBBuff         =   ( Q2 & !Q1 &  Q0) #
                          (!Q2 & !Q1 &  Q0);
 OB_Sel              =  !( Q2 & !Q1 & !Q0);
  end EBML;
```

```
module EBSR flag '-r2'
title 'External Bus Slave Register part, PAL 16R4
Dave Arpin and Bob Kaucic  Dec 4, 1985'
  ue3   device   'P16R4';

"P16R4"
  Clock      pin in ue3  1;
  _R         pin in ue3  2;
  NMFul      pin in ue3  3;
  IBBFul     pin in ue3  4;
  INFul      pin in ue3  5;
  Grant      pin in ue3  6;
  DataRdy_MS pin in ue3  7;

"Outputs"
  DataRdy_SM pin in ue3 19;
  _EBon      pin in ue3 18;
  Q2         pin in ue3 17;
  Q1         pin in ue3 16;
  Q0         pin in ue3 15;
  OBFul      pin in ue3 14;
  _SNDNMB    pin in ue3 13;
  _SNDIBB    pin in ue3 12;

"Pin 11 will be grounded to enable outputs"

CP = .C.;


"States"
 S0=^b111;
 S1=^b110;
 S2=^b101;
 S3=^b100;
 S4=^b011;
 S5=^b010;
 S6=^b001;
 S7=^b000;

equations
ENABLE DataRdy_SM   = 1;
ENABLE _EBon        = 1;
ENABLE _SNDNMB      = 1;
ENABLE _SNDIBB      = 1;

STATE_DIAGRAM in ue3 [Q2,Q1,Q0]

 state S0 : !_SNDNMB = !DataRdy_MS & !OBFul & NMFul;
            !_SNDIBB = !DataRdy_MS & !OBFul & !NMFul
                                   & IBBFul;
            OBFul   := OBFul;
```

```
                    !_EBon   = !DataRdy_MS & Grant & OBFul;
                    DataRdy_SM = !DataRdy_MS & Grant & OBFul;

                    if !_R then S6
                    else
                    if !DataRdy_MS & !OBFul & NMFul then S1
                    else
                    if !DataRdy_MS & !OBFul & !NMFul
                       & IBBFul then S2
                    else
                    if !DataRdy_MS & Grant & OBFul then S3
                    else
                    if DataRdy_MS then S4
                    else S0;

        state S1 : !_SNDNMB         = 1;
                   OBFul           := 1;
                   !_SNDIBB         = 0;
                   DataRdy_SM       = 0;
                   !_EBon           = 0;
                   if !_R then S6
                   else
                   if !DataRdy_MS then S0
                   else S4;

        state S2 : !_SNDIBB         = 1;
                   OBFul           := 1;
                   !_SNDNMB         = 0;
                   DataRdy_SM       = 0;
                   !_EBon           = 0;
                   if !_R then S6
                   else
                   if !DataRdy_MS then S0
                   else S4;

        state S3 : !_EBon           = 1;
                   DataRdy_SM       = 1;
                   OBFul           := 0;
                   !_SNDIBB         = 0;
                   !_SNDNMB         = 0;
                   if !_R then S6
                   else S0;

        state S4 : OBFul           := OBFul;
                   !_SNDIBB         = 0;
                   !_SNDNMB         = 0;
                   DataRdy_SM       = 0;
                   !_EBon           = 0;
                   if !_R then S6
                   else
                   if !DataRdy_MS then S4
```

```
           else S0;

"Take care of power up conditions"

state S5 : OBFul          := 0;
           !_SNDIBB        = 0;
           !_SNDNMB        = 0;
           DataRdy_SM      = 0;
           !_EBon          = 0;
           GOTO S6;


state S6 : OBFul          := 0;
           !_SNDIBB        = 0;
           !_SNDNMB        = 0;
           !_EBon          = 0;
           DataRdy_SM      = 0;
           if _R then S0
           else S6;

state S7 : OBFul          := 0;
           !_SNDIBB        = 0;
           !_SNDNMB        = 0;
           !_EBon          = 0;
           DataRdy_SM      = 0;
           GOTO S6;

test_vectors in ue3
([_R, Clock, Q2, Q1, Q0] -> [Q2,Q1,Q0])
 [0 ,  CP  ,  1,  1,  1] -> S6;
 [0 ,  CP  ,  0,  0,  1] -> S6;
 [1 ,  CP  ,  0,  0,  1] -> S0;

end EBSR;
```

```
module EBSL flag '-r2'
title 'Written by Bob Kaucic'
 ue4    device   'P16L8';

"P16L8"
AUFull           pin in ue4  1;
AUOld            pin in ue4  2;
AMDone           pin in ue4  3;
_Needed_Here     pin in ue4  4;
RTFull           pin in ue4  5;
RTOld            pin in ue4  6;
RCUDone          pin in ue4  7;
Send_On          pin in ue4  8;
DataRdy_MS       pin in ue4  9;
Q0               pin in ue4 11;
Q1               pin in ue4 13;
Q2               pin in ue4 14;

_Clr_INFul       pin in ue4 12;
Set_INFul        pin in ue4 15;
Clock_OB         pin in ue4 16;
_Clr_NMBuff      pin in ue4 17;
_Clr_IBBuff      pin in ue4 18;
OB_Sel           pin in ue4 19;  "1 is upper bits"

equations in ue4
ENABLE OB_Sel      = 1;
ENABLE _Clr_IBBuff = 1;
ENABLE _Clr_NMBuff = 1;
ENABLE Clock_OB    = 1;
ENABLE Set_INFul   = 1;
ENABLE _Clr_INFul  = 1;

ENABLE Q2         = 0;
ENABLE Q1         = 0;

!_Clr_INFul          =   ((AUFull & !AUOld)
                           # (AMDone & _Needed_Here)) &
                         ((RTFull & !RTOld)
                           # (RCUDone & !Send_On))
                         # (!Q2 & !Q1 &  Q0);
 Set_INFul           =   (!Q2 &  Q1 &  Q0 & DataRdy_MS);
 Clock_OB            =  (( Q2 &  Q1 & !Q0) #
                         ( Q2 & !Q1 &  Q0));
!_Clr_NMBuff         =   ( Q2 &  Q1 & !Q0) #
                         (!Q2 & !Q1 &  Q0);
!_Clr_IBBuff         =   ( Q2 & !Q1 &  Q0) #
                         (!Q2 & !Q1 &  Q0);
 OB_Sel              =  !( Q2 & !Q1 & !Q0);

end EBSL;
```

```
module AUInt
title 'AU Interface Controller PAL
Dave Arpin  4 Dec 85'

 ual   device   'P16R4';

"Inputs"
 Clock          pin in ual  1;
 AU_DatRdy      pin in ual  2;
 SRDone         pin in ual  3;
 AMDone         pin in ual  4;
 _Needed_Here   pin in ual  5; "Active Low"
 RCUDone        pin in ual  6;
 Send_On        pin in ual  7;
 AUGranted      pin in ual  8;
 _R             pin in ual  9;

"Outputs"
 ReqAU          pin in ual 19;
 _CP_DatRdy     pin in ual 18;
 Q              pin in ual 17;
 AUFul          pin in ual 16;
 AUisAns        pin in ual 15;

EQUATIONS
ENABLE ReqAU = 1;

STATE_DIAGRAM IN ual [Q]

 state 0 :      ReqAU = AUFul & !AUisAns;
                ENABLE _CP_DatRdy = 0;
                !AUFul := (!AUFul & ((_Needed_Here # !AMDone)
                                     & !AU_DatRdy)
                        # AUFul & (SRDone & AUisAns)) # !_R;
                !AUisAns := (!AUisAns & !AU_DatRdy
                            # AUisAns & SRDone) # !_R;

                if !_R then 0
                else
                if AUFul & !AUisAns & AUGranted then 1
                else 0;

 state 1 :      ReqAU = 1;
                ENABLE _CP_DatRdy = 1;
                !_CP_DatRdy =1;
                AUFul := 0;
                AUisAns := 0;   "For power on reset"
                GOTO 0;

end AUInt;
```

```
module IBusCtrl   flag '-r2'
title 'Internal Bus Controller PAL
Dave Arpin    4 Dec 85'
                        uil   device   'P16R4';

"Inputs"
 Clock         pin in uil  1;
 RTBFul        pin in uil  2;
 IBBFul        pin in uil  3;
 _AddressedI   pin in uil  4;
 _IBGranted    pin in uil  5;
 RCUDone       pin in uil  6;
 Send_On       pin in uil  7;
 _R            pin in uil  8;

 _RNOT         pin in uil 11;

"I/O"
 _IDataRdy     pin in uil 19;
 _Ack          pin in uil 18;

"Outputs"
 Q0            pin in uil 17;
 Q1            pin in uil 16;
 _Clear_RTFul  pin in uil 15; "This is a clocked output"
 Clock_IBB     pin in uil 13;
 Clock_RTB     pin in uil 12;

"Req=RTBFul, so no separate output is needed for Req"
"Data Enable=Q0, so no separate output needed"

"States"
s0=^b11;
s1=^b10;
s2=^b01;
s3=^b00;

STATE_DIAGRAM IN uil [Q1,Q0]

 state s0 :    ENABLE _IDataRdy = 0;
               ENABLE _Ack = 0;
               Clock_IBB = 0;
               Clock_RTB = !RTBFul & RCUDone & Send_On;
               !_Clear_RTFul := !_R;

               if !_R then s3
               else
               if _AddressedI & RTBFul & !_IBGranted then s1
               else
               if !_AddressedI then s2
               else s0;
```

```
    state s1 :      ENABLE _IDataRdy = 1;
                    ENABLE _Ack = 0;
                    Clock_IBB = 0;
                    Clock_RTB = 0;
                    !_Clear_RTFul := !_Ack # !_R;
                    _IDataRdy = 0;

                    if !_R then s3
                    else
                    if !_Ack then s0
                    else s1;

    state s2 :      ENABLE _IDataRdy = 0;
                    ENABLE _Ack = 1;
                    Clock_IBB = !_IDataRdy & !IBBFul;
                    !_Ack = !_IDataRdy & !IBBFul;
                    Clock_RTB = !RTBFul & RCUDone & Send_On;
                    !_Clear_RTFul := !_R;

                    if !_R then s3
                    else
                    if !_IDataRdy & !IBBFul then s0
                    else s2;

     state s3:      ENABLE _IDataRdy = 0;
                    ENABLE _Ack      = 0;
                    Clock_IBB        = 0;
                    Clock_RTB        = 0;
                    !_Clear_RTFul   := 1;
                    if _R then s0
                    else s3;

  end IBusCtrl;
```

APPENDIX E - Prototype Software Listing

Software for the ECB to drive the prototype of the ParSOR processing element.

```
*TRAP 14 CONSTANTS FOR I/O, CONVERSION

OUTPUT   EQU     243      ;STRING OUTPUT, NO CRLF
OUT1CR   EQU     227      ;STRING OUTPUT + CRLF
PORTIN1  EQU     241      ;STRING INPUT, ECHOES CHRS, CRLF
GETNUMA  EQU     226      ;CONVERT ASCII HEX TO BINARY
GETNUMD  EQU     225      ;CONVERT ASCII DECIMAL TO BINARY
PNT2HX   EQU     233      ;CONVERT HEX TO ASCII HEX
TUTOR    EQU     228      ;RETURN TO TUTOR

*PI/T REGISTER ADDRESSES

PGCR     EQU     $10001             ;PORT GENERAL CONTROL
PADDR    EQU     $10005             ;PORT A DATA DIRECTION
PBDDR    EQU     $10007             ;PORT B DATA DIRECTION
PACR     EQU     $1000D             ;PORT A CONTROL
PBCR     EQU     $1000F             ;PORT B CONTROL
PADR     EQU     $10011             ;PORT A DATA
PBDR     EQU     $10013             ;PORT B DATA

*8255 REGISTER ADDRESSES

PORTA    EQU     $7FF01
PORTB    EQU     $7FF03
PORTC    EQU     $7FF05
CNTRL    EQU     $7FF07

*8255 CONTROL BYTE DEFINITIONS

MOSO     EQU     $82      ;MASTER DATA OUT, SLAVE DATA OUT
MOSI     EQU     $83      ;MASTER DATA OUT, SLAVE DATA IN
MISO     EQU     $8A      ;MASTER DATA IN, SLAVE DATA OUT
MISI     EQU     $8B      ;MASTER DATA IN, SLAVE DATA IN


*TEST DATA BYTES
MSDATA   EQU     $A4
SMDATA   EQU     $BC
AUDATA   EQU     $A0
```

```
*************************************************************
*
*            CONTROL SIGNAL LOCATIONS IN 8255 AND PI/T PORTS
*________
*       |
*      7|----> M GRANT OUT
*      6|-
*    P 5|----> AU GRANT Y
*    O 4|----> AU GRANT X
*    R 3|-
*    T 2|-
*    A 1|----> S REQ OUT
*      0|----> S BUFF FULL OUT
*       |
*      7|<--->
* 8    6|<--->
*    P 5|<---> 'MASTER' DATA LINES
* 2  O 4|<--->
*    R  |
* 5  T 3|<--->
*    C 2|<--->
* 5    1|<---> 'SLAVE' DATA LINES
*      0|<--->
*       |
*      7|<---- M REQ IN
*      6|<---- M BUFF FULL IN
*    P 5|-
*    O 4|<---- AU REQ Y
*    R 3|<---- AU REQ X
*    T 2|-
*    B 1|-
*      0|<---- S GRANT IN
*       |
*_______|
*
*________
*       |
*      7|----> CLK EN
*      6|-
*    P 5|-
*    A 4|-
*    D 3|-
*    R 2|-
* P    1|-
* I    0|-
* /     |
* T    7|----> M DATA RDY MS
*      6|<---- M DATA RDY SM
*    P 5|-
*    B 4|----> AU DATA RDY OUT
*    D 3|<---- AU DATA RDY IN
*    R 2|-
*       1|----> S DATA RDY SM
*       0|<---- S DATA RDY MS
*________|
*
************************************************************

        ORG     $2000    ;STARTING ADDRESS OF CODE
        OPT     FRS      ;FORWARD REFERENCE SHORT (ABSOLUTE)
        OPT     BRS      ;FORWARD BRANCH SHORT

*MAIN PROG GOES HERE

* SEND SEVERAL SUCCESSIVE MESSAGES TO MASTER CP

        MOVE.B  #SMDATA,SOUTB    ;DATA TO OUTPUT BUFFER
        MOVE.L  #6,D0            ;# MSGS IN D0
LPA4    BSR.L   SENDSM
        BTST.B  #7,PORTB     ;CHECK FOR M REQ IN
        BEQ     NOREQ
        BSR.L   RECSM        ;IF THERE IS ONE, RECEIVE DATA
NOREQ   DBF     D0,LPA4      ;LOOP TILL ALL MSGS SENT
        MOVE.B  #TUTOR,D7    ;THEN RETURN TO SYSTEM
        TRAP    #14
*
* SEND MESSAGES TO MASTER AND SLAVE AT THE SAME TIME,
* THEN READ RESULTS COMING OUT.
*
SIMUL   CLR.B   D0           ;CLEAR BYTE USED FOR STATUS
        MOVE.B  D0,RDST      ;  "
        MOVE.B  #SMDATA,SOUTB ;DATA FOR SLAVE TO SEND
        MOVE.B  #MSDATA,MOUTB
        BSET.B  #1,PORTA     ;ASSERT REQUEST, S TO M
        BTST.B  #7,PORTB     ;SEE IF SLAVE WANTS TO SEND
        BEQ     WAITGR       ;IF NOT, WAIT FOR GRANT
        BSR.L   RECSM        ;IF SLAVE HAS DATA, RCV IT
WAITGR  BSR.L   CLKP         ;SEND CLOCK PULSE TO BOARD
        BTST.B  #0,PORTB     ;SEE IF GRANT SET
        BEQ     WAITGR
        MOVE.B  #MOSO,CNTRL ;RDY. SET BOTH PORTS TO OUTPUT
        MOVE.B  MOUTB,D7     ;MOVE MASTER DATA TO PORT (HI
        BSR.L   STCHI        ; NIBBLE)
        MOVE.B  SOUTB,D7     ;MOVE SLAVE DATA TO PORT (HI)
        ROR.B   #4,D7
        BSR.L   STCLO
        BSET.B  #7,PBDR      ;SET M DATA READY MS
        BSET.B  #1,PBDR      ;SET S DATA READY SM
        BSR.L   CLKP
        BCLR.B  #7,PBDR      ;CLEAR DATA READY
        BCLR.B  #1,PBDR
```

```
         MOVE.B   MOUTB,D7       ;SEND LOW NIBBLE TO PORTS
         ROR.B    #4,D7
         BSR.L    STCHI
         MOVE.B   SOUTB,D7
         BSR.L    STCLO
         BSET.B   #7,PBDR
         BSET.B   #1,PBDR
         BSR.L    CLKP
         BCLR.B   #7,PBDR
         BCLR.B   #1,PBDR
         MOVE.B   #MISI,CNTRL    ;DONE SENDING, PORT -> INPUT
WAIT4S   BSR.L    CLKP           ;WAIT FOR DATA TO ARRIVE, READ
         BTST.B   #7,PORTB       ;CHECK FOR M REQ IN
         BNE      RDSLV          ;IF SO, READ DATA FROM SLAVE
WAIT4M   BTST.B   #0,PBDR        ;OTHERWISE, CK FOR S DATRDY MS
         BNE      RDMST          ;IF SET, READ DATA FROM MASTER
         CMPI.B   #$0F,RDST      ;NO DATRDY, SEE IF BOTH RCV'D
         BLT      WAIT4S         ;NO, SO CONTINUE WAITING
         BSR.L    CLKP           ;ONE MORE CLOCK TO CLEAR FLAGS
         MOVE.B   #TUTOR,D7      ;ALL DATA RCV'D, RET'N TO TUTOR
         TRAP     #14
RDSLV    BSET.B   #7,PORTA       ;SLAVE SET REQ, SO ASSERT GRANT
         BTST.B   #6,PBDR        ;CHECK FOR M DATA READY SM
         BEQ      WAIT4M         ;IF DATA NOT RDY CHECK MASTER
         MOVE.B   PORTC,D7       ;DATA READY - READ IT IN
         AND.B    #$F0,D7        ;MASK OFF LOWER HALF
                                 ;(IT IS MASTER DATA)
         BTST.B   #1,RDST        ;SEE IF HIGH NIBBLE WAS RCVD
         BNE      RDSLO          ;IF SO, READ THIS AS LOW NIBBLE
         MOVE.B   D7,MINB        ;THIS IS HIGH NIBBLE, STORE IT
         BSET.B   #1,RDST        ;SET FLAG SHOWING THIS IS DONE
         BRA      WAIT4M         ;AND DO MASTER READ
RDSLO    LSR.B    #4,D7          ;THIS IS LOW NIBBLE; SHIFT IN
         OR.B     D7,MINB        ;D7 & STORE IN LOW NIBBLE OF
                                 ; MASTER INPUT BUFFER
         BSET.B   #0,RDST        ;SET FLAG SHOWING THIS IS DONE,
         BRA      WAIT4M         ;THEN GO READ MASTER DATA
RDMST    MOVE.B   PORTC,D7       ;DRMS ON; READ DATA FROM MASTER
         AND.B    #$0F,D7        ;MASK OFF SLAVE DATA
         BTST.B   #3,RDST        ;SEE IF HIGH NIBBLE RECEIVED
         BNE      RDMLO          ;IF SO, GO READ AS LOW NIBBLE
         LSL.B    #4,D7          ;READ HIGH NIBBLE FROM MASTER
         MOVE.B   D7,SINB        ; TO SLAVE INPUT BUFFER
         BSET.B   #3,RDST        ;SET FLAG TO SHOW THIS IS DONE
         BRA      WAIT4S
RDMLO    OR.B     D7,SINB        ;READ IN LOW NIBBLE FROM MASTER
         BSET.B   #2,RDST
         BRA      WAIT4S         ;CONTINUE
```

```
* SEND SEVERAL SUCCESSIVE MESSAGES TO SLAVE
* TO TEST PIPELINING SPEED

         MOVE.B    #MSDATA,MOUTB
         MOVE.L    #6,D0
LPA3     BSR.L     SENDMS
         DBF       D0,LPA3
         MOVE.B    #TUTOR,D7
         TRAP      #14

         BSR     INIT                ;SENDS DATA TO AU
         MOVE.B  #AUDATA,AUOUTB
         BSR.L   SENDAU
         MOVE.B  #TUTOR,D7
         TRAP    #14

         BSR     INIT                ;OUTPUTS TEN CLOCK PULSES
         MOVE.L  #9,D7
LOOP     BSR     CLKP
         DBF     D7,LOOP
         MOVE.B  #TUTOR,D7
         TRAP    #14

*UTILITY SUBROUTINES

*FOR INITIALIZING THE PI/T AND 8255

INIT     MOVE.B  #0,PGCR             ;MODE 0, NO HANDSHAKING
         MOVE.B  #$A0,PACR           ;SUBMODE 1X
         MOVE.B  #$A0,PBCR           ; FOR A AND B
         MOVE.B  #$FF,PADDR          ;ALL OUTPUT
         MOVE.B  #$92,PBDDR          ;INPUT, OUTPUT PATTERN
         MOVE.B  #0,PADR             ;CLEAR OUTPUT LINES
         MOVE.B  #0,PBDR

         MOVE.B  #MISI,CNTRL         ;8255 MASTER IN SLAVE IN
         MOVE.B  #0,PORTA            ;CLEAR OUTPUT LINES
         BSR     CLKP              ;GET PAL OUT OF INIT STATE
         RTS

*******************************************************
*FOR TRIGGERING A CLOCK PULSE AND COUNTING CLOCK PULSES
********************************************************

CLKP     BSET.B  #7,PADR             ;SET BIT 7, PI/T PORT A
         BCLR    #7,PADR             ;CLEAR BIT 7
         ADDQ.B  #1,CLKCNT           ;ADD ONE TO CLOCK COUNTER
         NOP                         ;BCS  CNTOV GOES HERE
         RTS
```

```
*FOR DISPLAYING ERROR MESSAGE IF COUNTER OVERFLOW

CNTOV     LEA      EROV,A5     ;LOAD POINTER WITH START OF MSG
          LEA      ENDMESS,A6  ;LOAD POINTER WITH END OF MSG
          MOVE.B   #OUT1CR,D7         ;OUTPUT MESSAGE
          TRAP     #14
          RTS

*SEND DATA FROM MASTER TO SLAVE, DATA MUST BE STORED IN
*MOUTB BEFORE CALLING

SENDMS    MOVE.L   D7,-(SP)        ;SAVE D7 ON STACK
          MOVE.B   #MOSI,CNTRL     ;MASTER DATA OUT
CHKBF     BTST.B   #6,PORTB        ;TEST FOR BUFF FULL INPUT
          BEQ      XMS             ;IF NOT ASSERTED, GO TRANSMIT
          BSR      CLKP            ;ELSE OUTPUT CLOCK PULSE
          BRA      CHKBF           ;GO CHECK BUFF FULL AGAIN
XMS       MOVE.B   MOUTB,D7        ;CONTENTS OF OUT BUFF INTO D7
          BSR      STCHI           ;HIGH NIBBLE OF D7 TO PORT C
          BSET.B   #7,PBDR         ;ASSERT M DATA RDY MS
          BSR      CLKP            ;CLOCK PULSE
          BCLR.B   #7,PBDR         ;NEGATE M DATA RDY MS
          MOVE.B   MOUTB,D7        ;RESTORE D7, ALTERED BY STCHI
          ROR.B    #4,D7           ;SWAP HIGH & LOW NIBBLES
          BSR      STCHI           ;LOW NIBBLE OF D7 TO PORT C
          BSET.B   #7,PBDR         ;ASSERT M DATA RDY MS
          BSR      CLKP            ;CLOCK PULSE
          BCLR.B   #7,PBDR         ;NEGATE M DATA RDY MS
          BSR      CLKP            ;CLOCK PULSE
          MOVE.B   #MISI,CNTRL     ;TRI-STATE DATA LINES
          MOVE.L   (SP)+,D7        ;RESTORE D7
          RTS                      ;RETURN

* PLACE DATA IN UPPER NIBBLE OF PORTC WITHOUT DISTURBING
* DATA IN LOWER NIBBLE
STCHI     MOVE.L     D6,-(SP)      ;SAVE D6
          MOVE.B     D7,D6
          AND.B      #$F0,D6       ;CLEAR LOW NIBBLE OF DATA
          MOVE.B     PORTC,D7      ;CURRENT PORTC CONTENTS TO D7
          AND.B      #$0F,D7       ;CLEAR HIGH NIBBLE (OLD)
          OR.B       D6,D7         ;PUT NEW DATA IN HIGH NIBBLE
          MOVE.B     D7,PORTC      ;PLACE REVISED VAL. IN PORTC
          MOVE.L     (SP)+,D6      ;RESTORE D6
          RTS

*SEND DATA FROM SLAVE TO MASTER, DATA IN SOUTB

SENDSM    MOVE.L   D7,-(SP)        ;SAVE D7 ON STACK
          BSET.B   #1,PORTA        ;ASSERT S REQ OUT
```

```
CHKDRMS  BSR      CLKP            ;CLOCK PULSE
         BTST.B   #0,PBDR         ;TEST FOR S DATA RDY MS
         BEQ      CHKG
         BSR.L    RECMS           ;IF ASSERTED, GO RECEIVE DATA
CHKG     BTST.B   #0,PORTB        ;CHECK FOR GRANT
         BNE      XSM             ;IF ASSERTED, GO SEND DATA
         BRA      CHKDRMS         ;RECHECK S DATRDY MS & GRANT
XSM      MOVE.B   SOUTB,D7        ;CONTENTS OF OUTBUF INTO D7
         ROR.B    #4,D7           ;SWAP HIGH & LOW NIBBLES
         MOVE.B   #MISO,CNTRL     ;SLAVE DATA OUT
         BSET.B   #1,PORTA        ;RE-ASSERT S REQ OUT
         BSR      STCLO           ;LOW NIBBLE OF DATA TO PORTC
         BSET.B   #1,PBDR         ;ASSERT S DATA RDY SM
         BSR      CLKP            ;CLOCK PULSE
         BCLR.B   #1,PBDR         ;NEGATE S DATA RDY SM
         MOVE.B   SOUTB,D7        ;RESTORE D7, ALTERED BY STCLO
         BSR      STCLO           ;TRANSMIT LOW NIBBLE
         BSET.B   #1,PBDR         ;ASSERT S DATA RDY SM
         BSR      CLKP            ;CLOCK PULSE
         BCLR.B   #1,PBDR         ;NEGATE S DATA RDY SM
         BSR      CLKP            ;CLOCK PULSE
         MOVE.B   #MISI,CNTRL     ;TRI-STATE DATA LINES
         MOVE.L   (SP)+,D7        ;RESTORE D7
         RTS                      ;RETURN

* PLACE DATA FROM LOWER NIBBLE OF D7 IN PORTC
* WITHOUT DISTURBING DATA IN UPPER NIBBLE.

STCLO    MOVE.L     D6,-(SP)        ;SAVE D6
         MOVE.B     D7,D6
         AND.B      #$0F,D6         ;CLEAR UPPER NIBBLE OF DATA
         MOVE.B     PORTC,D7        ;OLD PORTC CONTENTS TO D7
         AND.B      #$F0,D7         ;CLEAR LOW NIBBLE
         OR.B       D6,D7           ;COMBINE OLD AND NEW DATA
         MOVE.B     D7,PORTC        ;PUT NEW DATA IN PORTC
         MOVE.L     (SP)+,D6        ;RESTORE D6
         RTS

*RECEIVE DATA FROM MASTER TO SLAVE, DATA RETURNED IN SINB
*CALLED WHEN 'S DATA RDY MS' ASSERTED

RECMS    MOVE.L   D7,-(SP)        ;SAVE D7
         MOVE.B   PORTC,D7        ;INPUT HIGH NIBBLE OF DATA
         LSL.B    #4,D7           ;SHIFT INTO HIGH HALF OF D7
         MOVE.B   D7,SINB         ;HIGH NIBBLE TO INPUT BUFFER
         BSR      CLKP            ;CLOCK PULSE
         MOVE.B   PORTC,D7        ;INPUT LOW NIBBLE OF DATA
         AND.B    #$0F,D7         ;MASK OFF UPPER NIBBLE
         OR.B     D7,SINB         ;LOW NIBBLE TO INPUT BUFFER
         BSR      CLKP            ;CLOCK PULSE
         MOVE.L   (SP)+,D7        ;RESTORE D7
         RTS

*RECEIVE DATA FROM SLAVE TO MASTER, DATA RETURNED IN MINB
*CALLED WHEN 'M REQ IN' SET & MASTER HAS NO DATA TO SEND

RECSM    MOVE.L  D7,-(SP)       ;SAVE D7
         BSET.B  #7,PORTA       ;ASSERT M GRANT OUT
         BSR     CLKP           ;CLOCK PULSE
CHKDR    BTST.B  #6,PBDR        ;CHECK FOR M DATA RDY SM
         BNE     RSM            ;IF ASSERTED, RECEIVE DATA
         BSR     CLKP           ;ELSE CLOCK PULSE
         BRA     CHKDR          ; AND GO CHECK AGAIN
RSM      MOVE.B  PORTC,D7       ;INPUT HIGH NIBBLE OF DATA
         AND.B   #$F0,D7        ;MASK OFF LOW GARBAGE NIBBLE
         MOVE.B  D7,MINB        ;HIGH NIBBLE TO INPUT BUFFER
         BSR     CLKP           ;CLOCK PULSE
         MOVE.B  PORTC,D7       ;INPUT LOW NIBBLE OF DATA
         LSR.B   #4,D7          ;SHIFT INTO LOW HALF OF D7
         OR.B    D7,MINB        ;LOW NIBBLE TO INPUT BUFFER
         BSR     CLKP           ;CLOCK PULSE
         MOVE.L  (SP)+,D7       ;RESTORE D7
         RTS


SENDAU   BSET.B  #4,PBDR             ;ASSERT AU DATA RDY OUT
         BSR     CLKP                ;CLOCK PULSE
         BCLR.B  #4,PBDR             ;NEGATE AU DATA RDY OUT
         BSR     CLKP                ;CLOCK PULSE
         RTS

AURECEIV BTST.B  #3,PORTB            ;CHECK AU REQ X
         BNE     XRECEIVE            ;GRANT X
         BTST.B  #4,PORTB            ;CHECK AU REQ Y
         BNE     YRECEIVE            ;GRANT Y
         BSR     CLKP                ;CLOCK PULSE
         BRA     AURECEIV            ;WAIT FOR REQUEST

XRECEIVE BSET.B  #4,PORTA            ;SET X GRANT
WAITDRIN BSR     CLKP                ;CLOCK PULSE
         BTST.B  #3,PBDR             ;CHECK AU DATA RDY IN
         BNE     WAITDRIN
         BCLR.B  #4,PORTA            ;NEGATE X GRANT
         BRA     DONE

YRECEIVE BSET.B  #5,PORTA            ;ASSERT Y GRANT
YWAITDR  BSR     CLKP                ;CLOCK PULSE
         BTST.B  #3,PBDR             ;CHECK AU DATA RDY IN
         BNE     YWAITDR             ;WAIT FOR AU DATA RDY IN
         BCLR.B  #5,PORTA            ;NEGATE Y GRANT
DONE     RTS
```

```
CLKCNT   DS.B    1          ;STORAGE SPACE FOR CLOCK COUNTER
MINB     DS.B    1          ;MASTER AND SLAVE I/O BUFFERS
MOUTB    DS.B    1
SINB     DS.B    1
SOUTB    DS.B    1
RDST     DS.B    1
AUOUTB   DS.B    1
AUINB    DS.B    1
OUTBUF   DS.W    1


*MESSAGE STRINGS

EROV     DC.B    'WARNING: CLOCK COUNTER OVERFLOW'
ENDMESS  EQU     *
MSG1     DC.B    'THE MASTER SENT A '
ENDMSG1  EQU     *
MSG2     DC.B    'THE SLAVE RECEIVED A '
ENDMSG2  EQU     *
MSG3     DC.B    'THE SLAVE SENT A '
ENDMSG3  EQU     *
MSG4     DC.B    'THE MASTER RECEIVED A '
ENDMSG4  EQU     *

         END
```

VITA

David Arthur Arpin was born June 17, 1950 in Norwich, Connecticut. He graduated High School from Seattle Preparatory School, Seattle, Washington in 1969. He received the degree of Bachelor of Science in Electrical Engineering from the University of Washington, Seattle, Washington in 1973. He received the degree of Master of Science in Electrical Engineering from the Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, in 1977.

END
10-86
DTIC